Functional Data Structures and Algorithms

A Proof Assistant Approach

Tobias Nipkow (Ed.)

January 3, 2026

Preface

This book is an introduction to data structures and algorithms for functional languages, with a focus on proofs. It covers both functional correctness and running time analysis. It does so in a unified manner with inductive proofs about functional programs and their running time functions.

What sets this book apart from existing books on algorithms is that all proofs have been machine-checked, by the proof assistant Isabelle. That is, in addition to the text in the book, *which requires no knowledge of proof assistants!*, the Isabelle definitions and proofs are available online. Simply follow the links attached to chapter and section headings with a ↗ symbol. The structured nature of Isabelle proofs permits even novices to browse them and follow the high-level arguments.

This book is aimed at teachers and students (it has been classroom-tested for a number of years) but is also a reference work for programmers and researchers who are interested in the (verified!) details of some algorithm or proof.

Isabelle ↗

Isabelle [Nipkow et al. 2002, Paulson 1989, Wenzel 2002] is a proof assistant for the logic HOL (= Higher-Order Logic), which is why the system is often called Isabelle/HOL. HOL is a generalization of first-order logic: functions can be passed as parameters and returned as results, just as in functional programming, and they can be quantified over. Isabelle also supports a simple version of Haskell's type classes.

The main strength of proof assistants is their trustworthiness: all proofs are checked to be logically correct. Beyond trustworthiness, formal proofs can also clarify arguments, by exposing and explaining difficult steps. Most Isabelle users will confirm that their pen-and-paper proofs became clearer and less error-prone after they subjected themselves to the discipline of formal proof.

As emphasized above, the reader need not be familiar with Isabelle or HOL in order to read this book. However, to take full advantage of our proof assistant approach, readers are encouraged to learn how to write Isabelle definitions and proofs themselves — and to solve some of the exercises in this book. To this end we recommend the tutorial *Programming and Proving in Isabelle/HOL* [Nipkow], which is also Part I of the book *Concrete Semantics* [Nipkow and Klein 2014].

Prerequisites

We expect the reader to be familiar with

- the basics of discrete mathematics: propositional and first-order logic, sets and relations, proof principles including induction;
- a typed functional programming language like Haskell [Haskell], OCaml [OCaml] or Standard ML [Paulson 1996];
- simple inductive proofs about functional programs.

Under Development

This book is meant to grow. New chapters are meant to be added over time. The list of authors is meant to grow — *you* could become one of them!

Colour

For the quick orientation of the reader, definitions are displayed in coloured boxes:

These boxes display functional programs.

These boxes display auxiliary definitions.

From a logical point of view there is no difference between the two kinds of definitions except that auxiliary definitions need not be executable.

Acknowledgements We are obviously indebted to the books by Cormen et al. [2009] and Okasaki [1998]. We are similarly indebted to Makarius Wenzel, the long-time Isabelle architect. Fabian Huch co-taught the course based on this book multiple times. Jonas Stahl automated time function definitions. Lijun Chen, Nils Ole Harmsen, Magnus Myreen, Alex Nelson and Johannes Pohjola commented on or reported mistakes in preliminary versions of the book. We are very grateful to all of them.

Contents

1 Basics

Tobias Nipkow

In this chapter we describe the basic building blocks the book rests on.

Programs: The functional programming language we use is merely sketched because of its similarity with other well known functional languages.

Predefined types and notation: We introduce the basic predefined types and notations used in the book.

Inductive proofs: Although we do not explain proofs in general, we make an exception for certain inductive proofs.

Running time: We explain how we model running time by step counting functions.

1.1 Programs

The programs in this book are written in Isabelle's functional programming language which provides recursive algebraic data types (keyword: **datatype**), recursive function definitions and **let**, **if** and **case** expressions. The language is sufficiently close to a number of similar typed functional languages (SML [Paulson 1996], OCaml [OCaml], Haskell [Haskell]) to obviate the need for a detailed explanation. Moreover, Isabelle can generate SML, OCaml, Haskell and Scala code [Haftmann b]. What distinguishes Isabelle's functional language from ordinary programming languages is that all functions in Isabelle must terminate. Termination must be proved. For most of the functions in this book, termination is not difficult to see and Isabelle can prove it automatically. (For details on termination proofs, consult the function definition tutorial [Krauss].)

Isabelle's functional language is pure logic. All language elements have precise definitions. However, this book is about algorithms, not programming language semantics. A functional programmer's intuition suffices for reading it. (If you want to know more about the logical basis of recursive data types, recursive functions and code generation: see [Berghofer and Wenzel 1999, Haftmann and Nipkow 2010, Krauss 2006].)

A useful bit of notation: any infix operator can be turned into a function by enclosing it in parentheses, e.g. $(+)$.

1.2 Types

Type variables are denoted by $'a$, $'b$, etc. The function type arrow is $\Rightarrow$. Type constructor names follow their argument types, e.g. $'a\ list$. The notation $t :: \tau$ means that term t has type τ. The following types are predefined.

Booleans Type *bool* comes with the constants *True* and *False* and the usual operations. We mostly write $=$ instead of $\longleftrightarrow$.

Numbers There are three numeric types: the natural numbers *nat* (0, 1, . . .), the integers *int* and the real numbers *real*. They correspond to the mathematical sets $\mathbb{N}$, $\mathbb{Z}$ and $\mathbb{R}$ and not to any machine arithmetic. All three types come with the usual (overloaded) operations.

Sets The type $'a\ set$ of sets (finite and infinite) over type $'a$ comes with the standard mathematical operations. The minus sign "$-$", unary or binary, can denote set complement or difference.

Lists The type $'a\ list$ of lists whose elements are of type $'a$ is a recursive data type:

```
datatype 'a list = Nil | Cons 'a ('a list)
```

Constant *Nil* represents the empty list and *Cons* $x\ xs$ represents the list with first element x, the **head**, and rest list xs, the **tail**. The following syntactic sugar is sprinkled on top:

$$\begin{aligned} [] &\equiv \mathit{Nil} \\ x \mathbin{\#} xs &\equiv \mathit{Cons}\ x\ xs \\ [x_1, \ldots, x_n] &\equiv x_1 \mathbin{\#} \ldots \mathbin{\#} x_n \mathbin{\#} [] \end{aligned}$$

The $\equiv$ symbol means that the left-hand side is merely an abbreviation of the right-hand side.

A library of predefined functions on lists is shown in Appendix A. The length of a list xs is denoted by $|xs|$.

Type $'a\ option$ The data type $'a\ option$ is defined as follows:

```
datatype 'a option = None | Some 'a
```

Pairs and Tuples Pairs are written $(a,\ b)$. Functions *fst* and *snd* select the first and second component of a pair: *fst* $(a,\ b) = a$ and *snd* $(a,\ b) = b$. The type *unit* contains only a single element $()$, the empty tuple.

Functions Functions $'a \Rightarrow \ 'b$ come with a predefined pointwise update operation, with its own notation:

$$f(a := b) = (\lambda x.\ \textbf{if}\ x = a\ \textbf{then}\ b\ \textbf{else}\ f\ x)$$

1.2.1 Pattern Matching

Functions are defined by equations and pattern matching, for example over lists. Natural numbers may also be used in pattern-matching definitions:

$$\mathit{fib}\ (n + 2) = \mathit{fib}\ (n + 1) + \mathit{fib}\ n$$

Occasionally we use an extension of pattern matching where patterns can be named. For example, the defining equation

$$f\ (x\ \#\ (y\ \#\ zs =: ys)) = ys\ @\ zs$$

introduces a variable ys on the left that stands for $y\ \#\ zs$ and can be referred to on the right. Logically it is just an abbreviation of

$$f\ (x\ \#\ y\ \#\ zs) = (\textbf{let}\ ys = y\ \#\ zs\ \textbf{in}\ ys\ @\ zs)$$

although it is suggestive of a more efficient interpretation. The general format is $\mathit{pattern} =: \mathit{variable}$.

1.2.2 Numeric Types and Coercions

The numeric types nat, int and $real$ are all distinct. Converting between them requires explicit **coercion** functions, in particular the **inclusion** functions $\mathit{int} :: nat \Rightarrow int$ and $\mathit{real} :: nat \Rightarrow real$ that do not lose any information (in contrast to coercions in the other direction). We do not show inclusions unless they make a difference. For example, $(m + n) :: real$, where m, $n :: nat$, is mathematically unambiguous because $\mathit{real}\ (m + n) = \mathit{real}\ m + \mathit{real}\ n$. On the other hand, $(m - n) :: real$ is ambiguous because $\mathit{real}\ (m - n) \neq \mathit{real}\ m - \mathit{real}\ n$ because $(0::nat) - n = 0$. In some cases we can also drop coercions that are not inclusions, e.g. $\mathit{nat} :: int \Rightarrow nat$, which coerces negative integers to 0: if we know that $i \geq 0$ then we can drop the nat in $\mathit{nat}\ i$.

We prefer type nat over type $real$ for ease of (Isabelle) proof. For example, for m, $n :: nat$ we prefer $m \leq 2^n$ over $\mathit{lg}\ m \leq n$, where lg is the binary logarithm.

1.2.3 Multisets

Informally, a **multiset** is a set where elements can occur multiple times. Multisets come with the following operations:

$$\begin{array}{rcl}
\{\!\!\} & :: & 'a\ \mathit{multiset} \\
(\in_\#) & :: & 'a \Rightarrow\ 'a\ \mathit{multiset} \Rightarrow \mathit{bool} \\
\mathsf{add\_mset} & :: & 'a \Rightarrow\ 'a\ \mathit{multiset} \Rightarrow\ 'a\ \mathit{multiset} \\
(+) & :: & 'a\ \mathit{multiset} \Rightarrow\ 'a\ \mathit{multiset} \Rightarrow\ 'a\ \mathit{multiset} \\
\mathsf{size} & :: & 'a\ \mathit{multiset} \Rightarrow \mathit{nat} \\
\mathsf{mset} & :: & 'a\ \mathit{list} \Rightarrow\ 'a\ \mathit{multiset} \\
\mathsf{set\_mset} & :: & 'a\ \mathit{multiset} \Rightarrow\ 'a\ \mathit{set} \\
\mathsf{image\_mset} & :: & ('a \Rightarrow\ 'b) \Rightarrow\ 'a\ \mathit{multiset} \Rightarrow\ 'b\ \mathit{multiset} \\
\mathsf{filter\_mset} & :: & ('a \Rightarrow \mathit{bool}) \Rightarrow\ 'a\ \mathit{multiset} \Rightarrow\ 'a\ \mathit{multiset} \\
\mathsf{sum\_mset} & :: & 'a\ \mathit{multiset} \Rightarrow\ 'a
\end{array}$$

Their meaning: $\{\!\!\}$ is the empty multiset; $(\in_\#)$ is the element test; *add_mset* adds an element to a multiset; $(+)$ is the sum of two multisets, where multiplicities of elements are added; *size* M, written $|M|$, is the number of elements in M, taking multiplicities into account; *mset* converts a list into a multiset by forgetting about the order of elements; *set_mset* converts a multiset into a set; *image_mset* applies a function to all elements of a multiset; *filter_mset* removes all elements from a multiset that do not satisfy the given predicate; *sum_mset* is the sum of the values of a multiset, the iteration of $(+)$ (taking multiplicity into account).

We use some additional suggestive syntax for some of these operations:

$$\begin{array}{rcl}
\{\!\{x \in_\# M \mid P\ x\}\!\} & \equiv & \mathsf{filter\_mset}\ P\ M \\
\{\!\{f\ x \mid x \in_\# M\}\!\} & \equiv & \mathsf{image\_mset}\ f\ M \\
\sum_\#\ M & \equiv & \mathsf{sum\_mset}\ M \\
\sum_{x \in_\# M} f\ x & \equiv & \mathsf{sum\_mset}\ (\mathsf{image\_mset}\ f\ M)
\end{array}$$

See Section C.3 in the appendix for an overview of such syntax.

1.3 Notation

We deviate from Isabelle's notation in favour of standard mathematics in a number of points:

- There is only one implication: $\Longrightarrow$ is printed as $\longrightarrow$ and $P \Longrightarrow Q \Longrightarrow R$ is printed as $P \wedge Q \longrightarrow R$.
- *length* xs is printed as $|xs|$.
- Multiplication is printed as $x \cdot y$.
- Exponentiation is uniformly printed as x^y.

- We sweep under the carpet that type nat is defined as a recursive data type: **datatype** $\mathit{nat} = 0 \mid \mathsf{Suc}\ \mathit{nat}$. In particular, constructor Suc is hidden: $\mathsf{Suc}^k\ 0$ is printed as k and $\mathsf{Suc}^k\ n$ (where n is not 0) is printed as $n + k$.
- Set comprehension syntax is the canonical $\{x \mid P\}$.

The reader who consults the Isabelle theories referred to in this book should be aware of these discrepancies.

1.4 Proofs

Proofs are the *raison d'être* of this book. Thus we present them in more detail than is customary in a book on algorithms. However, not all proofs:

- We omit proofs of simple properties of numbers, lists, sets and multisets, our predefined types. Obvious properties (e.g. $|xs\ @\ ys| = |xs| + |ys|$ or commutativity of $\cup$) are used implicitly without proof.
- With some exceptions, we only state properties if their proofs require induction, in which case we will say so, and we will always indicate which supporting properties were used.
- If a proposition is simply described as "inductive" or its proof is described by a phrase like "by an easy/automatic induction" it means that in the Isabelle proofs all cases of the induction were automatic, typically by simplification.

As a simple example of an easy induction consider the append function

$$
\begin{aligned}
&(@) :: {'a}\ \mathit{list} \Rightarrow {'a}\ \mathit{list} \Rightarrow {'a}\ \mathit{list}\\
&[]\ @\ ys = ys\\
&(x \# xs)\ @\ ys = x \# xs\ @\ ys
\end{aligned}
$$

and the proof of $(xs\ @\ ys)\ @\ zs = xs\ @\ ys\ @\ zs$ by structural induction on xs. (Note that $(@)$ associates to the right.) The base case is trivial by definition: $([]\ @\ ys)\ @\ zs = []\ @\ ys\ @\ zs$. The induction step is easy:

$$
\begin{aligned}
&(x \# xs\ @\ ys)\ @\ zs\\
&= x \# (xs\ @\ ys)\ @\ zs && \text{by definition of } (@)\\
&= x \# xs\ @\ ys\ @\ zs && \text{by IH}
\end{aligned}
$$

Note that **IH** stands for **Induction Hypothesis**, in this case $(xs\ @\ ys)\ @\ zs = xs\ @\ ys\ @\ zs$.

1.4.1 Computation Induction

Because most of our proofs are about recursive functions, most of them are by induction, and we say so explicitly. If we do not state explicitly what form the induction takes, it is by an obvious structural induction. The alternative and more general induction schema is **computation induction** where the induction follows the terminating computation, but from the bottom up. For example, the terminating recursive definition for *gcd* $:: nat \Rightarrow nat \Rightarrow nat$

$$\mathit{gcd}\ m\ n = (\textbf{if}\ n = 0\ \textbf{then}\ m\ \textbf{else}\ \mathit{gcd}\ n\ (m\ \mathrm{mod}\ n))$$

gives rise to the following induction schema:

If $(n \neq 0 \longrightarrow P\ n\ (m\ \mathrm{mod}\ n)) \longrightarrow P\ m\ n$ (for all m and n),
then $P\ m\ n$ (for all m and n).

In general, let $f :: \tau \Rightarrow \tau'$ be a terminating function of, for simplicity, one argument. Proving $P(x :: \tau)$ by induction on the computation of f means proving

$$P\ r_1 \wedge \ldots \wedge P\ r_n \longrightarrow P\ e$$

for every defining equation

$$f\ e = \ldots\ f\ r_1\ \ldots\ f\ r_n\ \ldots$$

where $f\ r_1, \ldots, f\ r_n$ are all the recursive calls. For simplicity we have ignored the **if** and **case** contexts that a recursive call $f\ r_i$ occurs in and that should be preconditions of the assumption $P\ r_i$ as in the *gcd* example. If the defining equations for f overlap, the above proof obligations are stronger than necessary.

1.5 Running Time

Our approach to reasoning about the **running time** of a function f is very simple: we explicitly define a function T_f such that $T_f\ x$ models the time the computation of $f\ x$ takes. More precisely, T_f counts the number of non-primitive function calls in the computation of f. It is not intended that T_f yields the exact running time but only that the running time of f is in $O(T_f)$.

Given a function $f :: \tau_1 \Rightarrow \ldots \Rightarrow \tau_n \Rightarrow \tau$ we define a **(running) time function** $T_f :: \tau_1 \Rightarrow \ldots \Rightarrow \tau_n \Rightarrow nat$ by translating every defining equation for f into a defining equation for T_f. The translation is defined by two functions: $\mathcal{E}$ translates defining equations for f to defining equations for T_f and $\mathcal{T}$ translates expressions that compute some value to expressions that computes the number of function calls. The unusual notation $\mathcal{E}[\![.]\!]$ and $\mathcal{T}[\![.]\!]$ emphasizes that they are not functions in the logic.

$$
\begin{aligned}
\mathcal{E}[\![f\,p_1\,\ldots\,p_n = e]\!] &= (T_f\ p_1\ \ldots\ p_n = \mathcal{T}[\![e]\!] + 1)\\
\mathcal{T}[\![f\ e_1\ \ldots\ e_n]\!] &= \mathcal{T}[\![e_1]\!] + \ldots + \mathcal{T}[\![e_n]\!] + T_f\ e_1\ \ldots\ e_n
\end{aligned}
\tag{1.1}
$$

This is the general idea. It requires some remarks and clarifications:

- This definition of T_f is an abstraction of a call-by-value semantics. Thus it is also correct for lazy evaluation but may be a very loose upper bound.
- Definition (1.1) is incomplete: if f is a variable or constructor function (e.g. *Nil* or *Cons*), then there is no defining equation and thus no T_f. Conceptually we define $T_f\ \ldots = 0$ if f is a variable, constructor function or predefined function on *bool* or numbers. That is, we count only user-defined function calls. This does not change $O(T_f)$ for user-defined functions f (see Discussion below).
- **if**, **case** and **let** are treated specially:

$$
\begin{aligned}
&\mathcal{T}[\![\textbf{if}\ b\ \textbf{then}\ e_1\ \textbf{else}\ e_2]\!]\\
&= \mathcal{T}[\![b]\!] + (\textbf{if}\ b\ \textbf{then}\ \mathcal{T}[\![e_1]\!]\ \textbf{else}\ \mathcal{T}[\![e_2]\!])\\
&\mathcal{T}[\![\textbf{case}\ e\ \textbf{of}\ p_1 \Rightarrow e_1 \mid \ldots \mid p_k \Rightarrow e_k]\!]\\
&= \mathcal{T}[\![e]\!] + (\textbf{case}\ e\ \textbf{of}\ p_1 \Rightarrow \mathcal{T}[\![e_1]\!] \mid \ldots \mid p_k \Rightarrow \mathcal{T}[\![e_k]\!])\\
&\mathcal{T}[\![\textbf{let}\ x = e_1\ \textbf{in}\ e_2]\!] = \mathcal{T}[\![e_1]\!] + (\textbf{let}\ x = e_1\ \textbf{in}\ \mathcal{T}[\![e_2]\!])
\end{aligned}
$$

- For simplicity we restrict ourselves to a first-order language above. Nevertheless we use a few basic higher-order functions like *map* in the book. Their running time functions are defined in Appendix B.1.

As an example consider the append function (@) defined above. The defining equations for $T_{append} :: {'a\ list} \Rightarrow {'a\ list} \Rightarrow nat$ are easily derived. The first equation translates like this:

$$
\begin{aligned}
&\mathcal{E}[\![[]\ @\ ys = ys]\!]\\
&=\ (T_{append}\ []\ ys = \mathcal{T}[\![ys]\!] + 1)\\
&=\ (T_{append}\ []\ ys = 1)
\end{aligned}
$$

The right-hand side of the second equation translates like this:

$$
\begin{aligned}
&\mathcal{T}[\![x\ \#\ xs\ @\ ys]\!]\\
&= \mathcal{T}[\![x]\!] + \mathcal{T}[\![xs\ @\ ys]\!] + T_{Cons}\ x\ (xs\ @\ ys)\\
&= 0 + (\mathcal{T}[\![xs]\!] + \mathcal{T}[\![ys]\!] + T_{append}\ xs\ ys) + 1\\
&= 0 + (0 + 0 + T_{append}\ xs\ ys) + 1
\end{aligned}
$$

Thus the two defining equations for T_{append} are

$$T_{\mathit{append}}\ []\ ys = 1$$
$$T_{\mathit{append}}\ (x\ \#\ xs)\ ys = T_{\mathit{append}}\ xs\ ys + 1$$

As a final simplification, we drop the $+1$ in the time functions for non-recursive functions (think inlining). In that case $\mathcal{E}\llbracket f\,x_1\ \dots\ x_n = e\rrbracket\ =\ (T_f\ x_1\ \dots\ x_n = \mathcal{T}\llbracket e\rrbracket)$. Again, this does not change $O(T_f)$ (except in the trivial case where $\mathcal{T}\llbracket e\rrbracket = 0$).

In the main body of the book we initially show the definition of each T_f. Once the principles above have been exemplified sufficiently, the time functions are relegated to Appendix B.

The definition of T_f from the definition of f has been automated in Isabelle.

1.5.1 Example: List Reversal

This section exemplifies not just the definition of time functions but also their analysis. The standard list reversal function *rev* is defined in Appendix A. This is the corresponding time function:

$$T_{\mathit{rev}} :: {'a}\ \mathit{list} \Rightarrow \mathit{nat}$$
$$T_{\mathit{rev}}\ [] = 1$$
$$T_{\mathit{rev}}\ (x\ \#\ xs) = T_{\mathit{rev}}\ xs + T_{\mathit{append}}\ (\mathit{rev}\ xs)\ [x] + 1$$

A simple induction shows $T_{\mathit{append}}\ xs\ ys = |xs| + 1$. The precise formula for T_{rev} is less immediately obvious (exercise!) but an upper bound is easy to guess and verify by induction:

$$T_{\mathit{rev}}\ xs \le (|xs| + 1)^2$$

We will frequently prove upper bounds only.

Of course one can also reverse a list in linear time:

$$\mathit{itrev} :: {'a}\ \mathit{list} \Rightarrow {'a}\ \mathit{list} \Rightarrow {'a}\ \mathit{list}$$
$$\mathit{itrev}\ []\ ys = ys$$
$$\mathit{itrev}\ (x\ \#\ xs)\ ys = \mathit{itrev}\ xs\ (x\ \#\ ys)$$

$$T_{\mathit{itrev}} :: {'a}\ \mathit{list} \Rightarrow {'a}\ \mathit{list} \Rightarrow \mathit{nat}$$
$$T_{\mathit{itrev}}\ []\ \_ = 1$$
$$T_{\mathit{itrev}}\ (x\ \#\ xs)\ ys = T_{\mathit{itrev}}\ xs\ (x\ \#\ ys) + 1$$

Function *itrev* has linear running time: $T_{itrev}\ xs\ ys = |xs| + 1$. A simple induction yields *itrev* $xs\ ys =$ *rev* $xs\ @\ ys$. Thus *itrev* implements *rev*: *rev* $xs =$ *itrev* $xs\ []$.

1.5.2 Discussion

Analysing the running time of a program requires a precise cost model. For imperative programs the standard model is the Random Access Machine (RAM), where each instruction takes one time unit. For functional programs a standard measure is the number of function calls. We follow Sands [1990, 1995] by counting only non-primitive function calls. One could also count variable accesses, primitive and constructor function calls. This would not change $O(T_f)$ because it would only add a constant to each defining equation for T_f. However, it would make reasoning about T_f more tedious.

A full proof that the execution time of our functional programs is in $O(T_f)$ on some actual software and hardware is a major undertaking: one would need to formalize the full stack of compiler, runtime system and hardware. We do not offer such a proof. Thus our formalization of "time" should be seen as conditional: given a stack that satisfies our basic assumptions in the definition of $\mathcal{E}$ and $\mathcal{T}$, our analyses are correct for that stack. Below we argue that these assumptions are reasonable (on a RAM), provided we accept that both the address space and numbers have a fixed size and cannot grow arbitrarily. Of course this means that actual program execution may abort if the resources are exhausted.

To simplify our argument, we assume that $\mathcal{T}$ counts all function calls and variable accesses (which does not change $O(T_f)$, as we argued above). Thus our basic assumption is that function calls take constant time. This is reasonable (on a RAM) because we just need to allocate, initialize and later deallocate a stack frame of constant size. It is of constant size because all parameters are references or numbers and thus of fixed size. We also assumed that variable access takes constant time. This is a standard RAM assumption. Assuming that constructor functions take constant time is reasonable because the memory manager could simply employ a single reference to the first free memory cell and increment that with each constructor function call. Garbage collection complicates matters. In the worst case we have to assume that garbage collection is switched off, which simply exhausts memory more quickly. Finally we assume that operations on *bool* and numbers take constant time. The former is obvious, the latter follows from our assumption that we have fixed-size numbers.

In the end, we are less interested in a specific model of time and more in the principle that time (and other resources) can be analyzed just as formally as functional correctness once the ground rules (e.g. $\mathcal{T}$) have been established.

1.5.3 Asymptotic Notation

The above approach to running time analysis is nicely concrete and avoids the more sophisticated machinery of asymptotic notation, $O(.)$ and friends. Thus we have intentionally lowered the entry barrier to the book for readers who want to follow the Isabelle formalization: we require no familiarity with Isabelle's real analysis library and in particular with the existing formalization of and automation for asymptotic notation [Eberl 2017b]. Of course this comes at a price: one has to come up with and reason about somewhat arbitrary constants in the analysis of individual functions. Moreover, we seldom appeal to the **master theorem** [Cormen et al. 2009] (although Eberl [2017b] provides a generalized version) but prove solutions to recurrence relations correct by induction. Again, this is merely to reduce the required mathematical basis and to show that it can be done. In informal explanations, typically when considering inessential variations, we do use standard mathematical notation and write, for example, $O(n \lg n)$.

Part I

Sorting and Selection

2 Sorting

Tobias Nipkow and Christian Sternagel

In this chapter we define and verify the following sorting functions: insertion sort, quicksort, and three variations of merge sort. We also analyze their running times (except for quicksort, whose running time analysis is beyond the scope of this book).

Sorting involves an ordering. We assume such an ordering to be provided by comparison operators $\leq$ and $<$ defined on the underlying type.

Sortedness of lists is defined as follows:

$$\mathsf{sorted} :: ('a{::}\mathit{linorder})\ \mathit{list} \Rightarrow \mathit{bool}$$

$$\begin{aligned}&\mathsf{sorted}\ [] = \mathsf{True}\\ &\mathsf{sorted}\ (x \ \#\ ys) = ((\forall y{\in}\mathsf{set}\ ys.\ x \leq y) \wedge \mathsf{sorted}\ ys)\end{aligned}$$

That is, every element is $\leq$ to all elements to the right of it: the list is sorted in increasing order.

The notation $'a{::}\mathit{linorder}$ restricts the type variable $'a$ to linear orders, which means that sorted is only applicable if a binary predicate $(\leq) :: 'a \Rightarrow 'a \Rightarrow \mathit{bool}$ is defined and $(\leq)$ is a **linear order**, i.e. the following properties are satisfied:

reflexivity:	$x \leq x$
transitivity:	$x \leq y \wedge y \leq z \longrightarrow x \leq z$
antisymmetry:	$a \leq b \wedge b \leq a \longrightarrow a = b$
linearity/totality:	$x \leq y \vee y \leq x$

Moreover, the binary predicate $(<)$ must satisfy

$$x < y \ \longleftrightarrow\ x \leq y \wedge x \neq y.$$

On the numeric types *nat*, *int* and *real*, $(\leq)$ is a linear order.

Note that *linorder* is a specific predefined example of a **type class** [Haftmann a]. We will not explain type classes any further because we do not require the general concept. In fact, we will mostly not even show the *linorder* restriction in types: you can assume that if you see $\leq$ or $<$ on a generic type $'a$ in this book, $'a$ is implicitly restricted to *linorder*, unless we explicitly say otherwise.

2.1 Specification of Sorting Functions

A sorting function $\mathit{sort} :: 'a\ \mathit{list} \Rightarrow 'a\ \mathit{list}$ (where, as usual, $'a :: \mathit{linorder}$) must obviously satisfy the following property:

$$\mathsf{sorted}\ (\mathit{sort}\ \mathit{xs})$$

However, this is not enough — otherwise, $\mathit{nil\_sort}\ \mathit{xs} = []$ would be a correct sorting function. The set of elements in the output must be the same as in the input, and each element must occur the same number of times. This is most readily captured with a multiset (see Section 1.2.3). Thus the second property that a sorting function sort must satisfy is

$$\mathsf{mset}\ (\mathit{sort}\ \mathit{xs}) = \mathsf{mset}\ \mathit{xs}$$

where function *mset* converts a list into its corresponding multiset.

2.2 Insertion Sort

Insertion sort is well-known for its intellectual simplicity and computational inefficiency. Its simplicity makes it an ideal starting point for this book. Below, it is implemented by the function *insort* with the help of the auxiliary function *insort1* that inserts a single element into an already sorted list.

$$\mathsf{insort1} :: 'a \Rightarrow 'a\ \mathit{list} \Rightarrow 'a\ \mathit{list}$$

$$\begin{aligned}
&\mathsf{insort1}\ x\ [] = [x]\\
&\mathsf{insort1}\ x\ (y \mathbin{\#} \mathit{ys}) = (\textbf{if}\ x \le y\ \textbf{then}\ x \mathbin{\#} y \mathbin{\#} \mathit{ys}\ \textbf{else}\ y \mathbin{\#} \mathsf{insort1}\ x\ \mathit{ys})
\end{aligned}$$

$$\mathsf{insort} :: 'a\ \mathit{list} \Rightarrow 'a\ \mathit{list}$$

$$\begin{aligned}
&\mathsf{insort}\ [] = []\\
&\mathsf{insort}\ (x \mathbin{\#} \mathit{xs}) = \mathsf{insort1}\ x\ (\mathsf{insort}\ \mathit{xs})
\end{aligned}$$

2.2.1 Correctness

We start by proving the preservation of the multiset of elements:

$$\mathsf{mset}\ (\mathsf{insort1}\ x\ \mathit{xs}) = \{\!\{x\}\!\} + \mathsf{mset}\ \mathit{xs} \tag{2.1}$$

$$\mathsf{mset}\ (\mathsf{insort}\ \mathit{xs}) = \mathsf{mset}\ \mathit{xs} \tag{2.2}$$

Both properties are proved by induction; the proof of (2.2) requires (2.1).

Now we turn to sortedness. Because the definition of *sorted* involves *set*, it is frequently helpful to prove multiset preservation first (as we have done above) because that yields preservation of the set of elements. That is, from (2.1) we obtain:

$$\mathit{set}\ (\mathit{insort1}\ x\ xs) = \{x\} \cup \mathit{set}\ xs \tag{2.3}$$

Two inductions prove

$$\mathit{sorted}\ (\mathit{insort1}\ a\ xs) = \mathit{sorted}\ xs \tag{2.4}$$

$$\mathit{sorted}\ (\mathit{insort}\ xs) \tag{2.5}$$

where the proof of (2.4) uses (2.3) and the proof of (2.5) uses (2.4).

2.2.2 Running Time

These are the running time functions (according to Section 1.5):

$$
\begin{array}{l}
T_{\mathit{insort1}} :: \ 'a \Rightarrow \ 'a\ \mathit{list} \Rightarrow \mathit{nat} \\
T_{\mathit{insort1}}\ \_\ [] = 1 \\
T_{\mathit{insort1}}\ x\ (y\ \#\ ys) = (\textbf{if}\ x \le y\ \textbf{then}\ 0\ \textbf{else}\ T_{\mathit{insort1}}\ x\ ys) + 1 \\
\\
T_{\mathit{insort}} :: \ 'a\ \mathit{list} \Rightarrow \mathit{nat} \\
T_{\mathit{insort}}\ [] = 1 \\
T_{\mathit{insort}}\ (x\ \#\ xs) = T_{\mathit{insort}}\ xs + T_{\mathit{insort1}}\ x\ (\mathit{insort}\ xs) + 1
\end{array}
$$

A dismal quadratic upper bound for the running time of insertion sort is proved readily:

Lemma 2.1. $T_{\mathit{insort}}\ xs \le (|xs| + 1)^2$

Proof. The following properties are proved by induction on xs:

$$T_{\mathit{insort1}}\ x\ xs \le |xs| + 1 \tag{2.6}$$

$$|\mathit{insort1}\ x\ xs| = |xs| + 1 \tag{2.7}$$

$$|\mathit{insort}\ xs| = |xs| \tag{2.8}$$

The proof of (2.8) needs (2.7). The proof of $T_{\mathit{insort}}\ xs \le (|xs| + 1)^2$ is also by induction on xs. The base case is trivial. The induction step is easy:

$$
\begin{array}{ll}
T_{\mathit{insort}}\ (x\ \#\ xs) = T_{\mathit{insort}}\ xs + T_{\mathit{insort1}}\ x\ (\mathit{insort}\ xs) + 1 & \\
\le (|xs| + 1)^2 + T_{\mathit{insort1}}\ x\ (\mathit{insort}\ xs) + 1 & \text{by IH} \\
\le (|xs| + 1)^2 + |xs| + 1 + 1 & \text{using (2.6) and (2.8)} \\
\le (|x\ \#\ xs| + 1)^2 & \square
\end{array}
$$

Exercise 2.1 asks you to show that *insort* actually has quadratic running time on all lists $[n,\ n{-}1,\ \ldots,\ 0]$.

2.3 Quicksort

Quicksort [Hoare 1961] is a divide-and-conquer algorithm that sorts a list as follows: pick a **pivot** element from the list; partition the remaining list into those elements that are smaller and those that are greater than the pivot (equal elements can go into either sublist); sort these sublists recursively and append the results. A particularly simple version of this approach, where the first element is chosen as the pivot, and the equal elements are put into the second sublist, looks like this:

```
quicksort :: 'a list ⇒ 'a list
quicksort [] = []
quicksort (x # xs)
= quicksort (filter (λy. y < x) xs) @ [x] @ quicksort (filter (λy. y ≥ x) xs)
```

2.3.1 Correctness

Preservation of the multiset of elements

$$\mathit{mset}\ (\mathit{quicksort}\ xs) = \mathit{mset}\ xs \tag{2.9}$$

is proved by computation induction using these lemmas:

$$\mathit{mset}\ (\mathit{filter}\ P\ xs) = \mathit{filter\_mset}\ P\ (\mathit{mset}\ xs)$$

$$(\forall x.\ P\ x = (\neg\ Q\ x)) \longrightarrow \mathit{filter\_mset}\ P\ M + \mathit{filter\_mset}\ Q\ M = M$$

A second computation induction proves sortedness

$$\mathit{sorted}\ (\mathit{quicksort}\ xs)$$

using the lemmas

$$\mathit{sorted}\ (xs\ @\ ys) = (\mathit{sorted}\ xs \land \mathit{sorted}\ ys \land (\forall x{\in}\mathit{set}\ xs.\ \forall y{\in}\mathit{set}\ ys.\ x \le y))$$

$$\mathit{set}\ (\mathit{quicksort}\ xs) = \mathit{set}\ xs$$

where the latter one is an easy consequence of (2.9).

We do not analyze the running time of *quicksort*. It is well known that in the worst case it is quadratic (exercise!) but that the average-case running time (in a certain sense) is $O(n \lg n)$. If the pivot is chosen randomly instead of always choosing the first element, the *expected* running time is also $O(n \lg n)$. The necessary probabilistic analysis is beyond the scope of this text but can be found elsewhere [Eberl 2017a, Eberl et al. 2018].

2.4 Top-Down Merge Sort

Merge sort is another prime example of a divide-and-conquer algorithm, and one whose running time is guaranteed to be $O(n \lg n)$. We will consider three variants and start with the simplest one: split the list into two halves, sort the halves separately and merge the results.

```
merge :: 'a list ⇒ 'a list ⇒ 'a list
merge [] ys = ys
merge xs [] = xs
merge (x # xs) (y # ys)
= (if x ≤ y then x # merge xs (y # ys) else y # merge (x # xs) ys)

msort :: 'a list ⇒ 'a list
msort xs
= (let n = |xs|
   in if n ≤ 1 then xs
      else merge (msort (take (n div 2) xs)) (msort (drop (n div 2) xs)))
```

2.4.1 Correctness

We start off with multisets and sets of elements:

$$\mathit{mset}\ (\mathit{merge}\ xs\ ys) = \mathit{mset}\ xs + \mathit{mset}\ ys \tag{2.10}$$

$$\mathit{set}\ (\mathit{merge}\ xs\ ys) = \mathit{set}\ xs \cup \mathit{set}\ ys \tag{2.11}$$

Proposition (2.10) is proved by induction on the computation of *merge* and (2.11) is an easy consequence.

Lemma 2.2. $\mathit{mset}\ (\mathit{msort}\ xs) = \mathit{mset}\ xs$

Proof by induction on the computation of *msort*. Let $n = |xs|$. The base case ($n \le 1$) is trivial. Now assume $n > 1$ and let $ys = \mathit{take}\ (n \text{ div } 2)\ xs$ and $zs = \mathit{drop}\ (n \text{ div } 2)\ xs$.

$$\begin{aligned}
&\mathit{mset}\ (\mathit{msort}\ xs) = \mathit{mset}\ (\mathit{msort}\ ys) + \mathit{mset}\ (\mathit{msort}\ zs) && \text{by (2.10)}\\
&= \mathit{mset}\ ys + \mathit{mset}\ zs && \text{by IH}\\
&= \mathit{mset}\ (ys\ @\ zs)\\
&= \mathit{mset}\ xs && \square
\end{aligned}$$

Now we turn to sortedness. An induction on the computation of *merge*, using (2.11), yields

$$\mathit{sorted}\ (\mathit{merge}\ xs\ ys) = (\mathit{sorted}\ xs \wedge \mathit{sorted}\ ys) \tag{2.12}$$

Lemma 2.3. $\mathit{sorted}\ (\mathit{msort}\ xs)$

The proof is an easy induction on the computation of *msort*. The base case ($n \leq 1$) follows because every list of length ≤ 1 is sorted. The induction step follows with the help of (2.12).

2.4.2 Running Time

To simplify the analysis, and in line with the literature, we only count the number of comparisons:

```
$C_{merge}$ :: $'a\ list \Rightarrow\ 'a\ list \Rightarrow nat$
$C_{merge}$ [] _ = 0
$C_{merge}$ _ [] = 0
$C_{merge}$ $(x\ \#\ xs)\ (y\ \#\ ys)$
= 1 + (if $x \leq y$ then $C_{merge}$ $xs\ (y\ \#\ ys)$ else $C_{merge}$ $(x\ \#\ xs)\ ys$)

$C_{msort}$ :: $'a\ list \Rightarrow nat$
$C_{msort}$ $xs$
= (let $n = |xs|$;
       $ys$ = take $(n$ div $2)\ xs$;
       $zs$ = drop $(n$ div $2)\ xs$
   in if $n \leq 1$ then 0
      else $C_{msort}$ $ys$ + $C_{msort}$ $zs$ + $C_{merge}$ (msort $ys$) (msort $zs$))
```

By computation inductions we obtain:

$$|\mathit{merge}\ xs\ ys| = |xs| + |ys| \tag{2.13}$$

$$|\mathit{msort}\ xs| = |xs| \tag{2.14}$$

$$C_{merge}\ xs\ ys \leq |xs| + |ys| \tag{2.15}$$

where the proof of (2.14) uses (2.13).

To simplify technicalities, we prove the $n \lg n$ bound on the number of comparisons in *msort* only for $n = 2^k$, in which case the bound becomes $k \cdot 2^k$.

Lemma 2.4. $|xs| = 2^k \longrightarrow C_{msort}\ xs \leq k \cdot 2^k$

Proof by induction on k. The base case is trivial and we concentrate on the step. Let $n = |xs|$, $ys = \mathit{take}\ (n \text{ div } 2)\ xs$ and $zs = \mathit{drop}\ (n \text{ div } 2)\ xs$. The case $n \leq 1$ is trivial. Now assume $n > 1$.

$$\begin{aligned}
& C_{\mathit{msort}}\ xs \\
&= C_{\mathit{msort}}\ ys + C_{\mathit{msort}}\ zs + C_{\mathit{merge}}\ (\mathit{msort}\ ys)\ (\mathit{msort}\ zs) \\
&\le C_{\mathit{msort}}\ ys + C_{\mathit{msort}}\ zs + |ys| + |zs| && \text{using (2.15) and (2.14)} \\
&\le k \cdot 2^k + k \cdot 2^k + |ys| + |zs| && \text{by IH} \\
&= k \cdot 2^k + k \cdot 2^k + |xs| \\
&= (k + 1) \cdot 2^{k + 1} \qquad \text{by assumption } |xs| = 2^{k + 1} && \square
\end{aligned}$$

2.5 Bottom-Up Merge Sort

Bottom-up merge sort starts by turning the input $[x_1, \ldots, x_n]$ into the list $[[x_1], \ldots, [x_n]]$. Then it passes over this list of lists repeatedly, merging pairs of adjacent lists on every pass until at most one list is left.

$$\begin{aligned}
& \mathit{merge\_adj} :: \ 'a\ list\ list \Rightarrow\ 'a\ list\ list \\
& \mathit{merge\_adj}\ [] = [] \\
& \mathit{merge\_adj}\ [xs] = [xs] \\
& \mathit{merge\_adj}\ (xs \mathbin{\#} ys \mathbin{\#} zss) = \mathit{merge}\ xs\ ys \mathbin{\#} \mathit{merge\_adj}\ zss
\end{aligned}$$

$$\begin{aligned}
& \mathit{merge\_all} :: \ 'a\ list\ list \Rightarrow\ 'a\ list \\
& \mathit{merge\_all}\ [] = [] \\
& \mathit{merge\_all}\ [xs] = xs \\
& \mathit{merge\_all}\ xss = \mathit{merge\_all}\ (\mathit{merge\_adj}\ xss)
\end{aligned}$$

$$\begin{aligned}
& \mathit{msort\_bu} :: \ 'a\ list \Rightarrow\ 'a\ list \\
& \mathit{msort\_bu}\ xs = \mathit{merge\_all}\ (\mathit{map}\ (\lambda x.\ [x])\ xs)
\end{aligned}$$

Termination of *merge_all* relies on the fact that *merge_adj* halves the length of the list (rounding up). Computation induction proves

$$|\mathit{merge\_adj2}\ acc\ xs| = |acc| + (|xs| + 1) \ \mathrm{div}\ 2 \tag{2.16}$$

2.5.1 Correctness

We introduce the abbreviation *mset_mset* $::\ 'a\ list\ list \Rightarrow\ 'a\ multiset$:

$$\mathit{mset\_mset}\ xss \equiv \textstyle\sum_{\#}\ (\mathit{image\_mset}\ \mathit{mset}\ (\mathit{mset}\ xss))$$

These are the key properties of the functions involved:

$$\mathit{mset\_mset}\ (\mathit{merge\_adj2}\ acc\ xss) = \mathit{mset\_mset}\ acc + \mathit{mset\_mset}\ xss$$

$$\mathit{mset}\ (\mathit{merge\_all2}\ xss) = \mathit{mset\_mset}\ xss \tag{2.17}$$

$$\mathsf{mset}\ (\mathsf{msort\_bu}\ xs) = \mathsf{mset}\ xs$$

$$(\forall xs \in \mathsf{set}\ xss.\ \mathsf{sorted}\ xs) \longrightarrow (\forall xs \in \mathsf{set}\ (\mathsf{merge\_adj}\ xss).\ \mathsf{sorted}\ xs)$$

$$(\forall xs \in \mathsf{set}\ xss.\ \mathsf{sorted}\ xs) \longrightarrow \mathsf{sorted}\ (\mathsf{merge\_all}\ xss) \tag{2.18}$$

$$\mathsf{sorted}\ (\mathsf{msort\_bu}\ xs)$$

The third and the last proposition prove functional correctness of *msort_bu*. The proof of each proposition may use the preceding propositions and the propositions (2.10) and (2.12). The propositions about *merge_adj* and *merge_all* are proved by computation inductions.

2.5.2 Running Time

Again, we count only comparisons:

$$\mathsf{C}_{\mathsf{merge\_adj}} :: {}'a\ list\ list \Rightarrow nat$$

$$\mathsf{C}_{\mathsf{merge\_adj}}\ [] = 0$$

$$\mathsf{C}_{\mathsf{merge\_adj}}\ [\_] = 0$$

$$\mathsf{C}_{\mathsf{merge\_adj}}\ (xs \mathbin{\#} ys \mathbin{\#} zss) = \mathsf{C}_{\mathsf{merge}}\ xs\ ys + \mathsf{C}_{\mathsf{merge\_adj}}\ zss$$

$$\mathsf{C}_{\mathsf{merge\_all}} :: {}'a\ list\ list \Rightarrow nat$$

$$\mathsf{C}_{\mathsf{merge\_all}}\ [] = 0$$

$$\mathsf{C}_{\mathsf{merge\_all}}\ [\_] = 0$$

$$\mathsf{C}_{\mathsf{merge\_all}}\ xss = \mathsf{C}_{\mathsf{merge\_adj}}\ xss + \mathsf{C}_{\mathsf{merge\_all}}\ (\mathsf{merge\_adj}\ xss)$$

$$\mathsf{C}_{\mathsf{msort\_bu}} :: {}'a\ list \Rightarrow nat$$

$$\mathsf{C}_{\mathsf{msort\_bu}}\ xs = \mathsf{C}_{\mathsf{merge\_all}}\ (\mathsf{map}\ (\lambda x.\ [x])\ xs)$$

By simple computation inductions we obtain:

$$\mathsf{even}\ |xss| \wedge (\forall xs \in \mathsf{set}\ xss.\ |xs| = m) \longrightarrow$$

$$(\forall xs \in \mathsf{set}\ (\mathsf{merge\_adj}\ xss).\ |xs| = 2 \cdot m) \tag{2.19}$$

$$(\forall xs \in \mathsf{set}\ xss.\ |xs| = m) \longrightarrow \mathsf{C}_{\mathsf{merge\_adj}}\ xss \le m \cdot |xss| \tag{2.20}$$

using (2.13) for (2.19) and (2.15) for (2.20).

Lemma 2.5. $(\forall xs \in \mathsf{set}\ xss.\ |xs| = m) \wedge |xss| = 2^k \longrightarrow$
$\mathsf{C}_{\mathsf{merge\_all}}\ xss \le m \cdot k \cdot 2^k$

Proof by induction on the computation of *merge_all*. We concentrate on the nontrivial recursive case arising from the third equation. We assume $|xss| > 1$, $\forall xs \in \mathsf{set}\ xss.\ |xs| = m$ and $|xss| = 2^k$. Clearly $k \ge 1$ and thus $\mathsf{even}\ |xss|$. Thus (2.19) implies $\forall xs \in \mathsf{set}\ (\mathsf{merge\_adj}\ xss).\ |xs| = 2 \cdot m$. Also note

$$
\begin{aligned}
&|\mathit{merge\_adj}\ xss| \\
&= (|xss| + 1) \text{ div } 2 && \text{using (2.16)} \\
&= 2^{k - 1} && \text{using } |xss| = 2^k \text{ and } k \geq 1 \text{ by arithmetic}
\end{aligned}
$$

Let $yss = \mathit{merge\_adj}\ xss$. We can now prove the lemma:

$$
\begin{aligned}
&C_{\mathit{merge\_all}}\ xss = C_{\mathit{merge\_adj}}\ xss + C_{\mathit{merge\_all}}\ yss \\
&\leq m \cdot 2^k + C_{\mathit{merge\_all}}\ yss && \text{using } |xss| = 2^k \text{ and (2.20)} \\
&\leq m \cdot 2^k + 2 \cdot m \cdot (k - 1) \cdot 2^{k - 1} \\
&\qquad \text{by IH using } \forall xs{\in}\mathit{set}\ yss.\ |xs| = 2 \cdot m \text{ and } |yss| = 2^{k - 1} \\
&= m \cdot k \cdot 2^k && \square
\end{aligned}
$$

For $m = 1$ we obtain the same upper bound as for top-down merge sort in Lemma 2.4:

Corollary 2.6. $|xs| = 2^k \longrightarrow C_{\mathit{msort\_bu}}\ xs \leq k \cdot 2^k$

2.6 Natural Merge Sort ↗

A disadvantage of all the sorting functions we have seen so far (except insertion sort) is that even in the best case they do not improve upon the $n \lg n$ bound. For example, given the sorted input [1, 2, 3, 4, 5], *msort_bu* will, as a first step, create [[1], [2], [3], [4], [5]] and then merge this list of lists recursively.

A slight variation of bottom-up merge sort, sometimes referred to as **natural merge sort**, first partitions the input into its constituent ascending and descending subsequences (collectively referred to as **runs**) and only then starts merging. In the above example we would get *merge_all* [[1, 2, 3, 4, 5]], which returns immediately with the result [1, 2, 3, 4, 5]. Assuming that obtaining runs is of linear complexity, this yields a best-case performance that is linear in the number of list elements.

Function *runs* computes the initial list of lists; it is defined mutually recursively with *asc* and *desc*, which gather ascending and descending runs in accumulating parameters:

```
runs :: 'a list ⇒ 'a list list
runs (a # b # xs) = (if b < a then desc b [a] xs else asc b ((#) a) xs)
runs [x] = [[x]]
runs [] = []

asc :: 'a ⇒ ('a list ⇒ 'a list) ⇒ 'a list ⇒ 'a list list
asc a as (b # bs)
= (if ¬ b < a then asc b (as ∘ (#) a) bs else as [a] # runs (b # bs))
asc a as [] = [as [a]]
```

$\mathit{desc} :: {}'a \Rightarrow {}'a\ \mathit{list} \Rightarrow {}'a\ \mathit{list} \Rightarrow {}'a\ \mathit{list}\ \mathit{list}$

$\mathit{desc}\ a\ as\ (b \mathbin{\#} bs)$
$= (\textbf{if}\ b < a\ \textbf{then}\ \mathit{desc}\ b\ (a \mathbin{\#} as)\ bs\ \textbf{else}\ (a \mathbin{\#} as) \mathbin{\#} \mathit{runs}\ (b \mathbin{\#} bs))$
$\mathit{desc}\ a\ as\ [] = [a \mathbin{\#} as]$

Function *desc* needs to reverse the descending run it collects. Therefore a natural choice for the type of its accumulator as is *list*, since recursively prepending elements (using $(\#)$) ultimately yields a reversed list.

Function *asc* collects an ascending run and is slightly more complicated than *desc*. If we used lists, we could accumulate the elements similarly to *desc* but using $as\ @\ [a]$ instead of $a \mathbin{\#} as$. This would take quadratic time in the number of appended elements. Therefore the “standard” solution is to accumulate elements using $(\#)$ and to reverse the accumulator in linear time (as shown in Section 1.5.1) at the end. However, another interesting option (that yields better performance for some functional languages, like Haskell) is to use **difference lists**. This is the option we chose for *asc*.

In the functional programming world, difference lists are a well-known trick to append lists in constant time by representing lists as functions of type $'a\ \mathit{list} \Rightarrow {}'a\ \mathit{list}$. For difference lists, we have the following correspondences: empty list $[] \approx \lambda x.\ x$, singleton list $[x] \approx (\#)\ x$, and list append $xs\ @\ ys \approx xs \circ ys$ (taking constant time). Moreover, transforming a difference list xs into a normal list is as easy as $xs\ []$ (taking linear time).

Note that, due to the mutually recursive definitions of *runs*, *asc*, and *desc*, whenever we prove a property of *runs*, we simultaneously have to prove suitable properties of *asc* and *desc* using mutual induction.

Natural merge sort is the composition of *merge_all* and *runs*:

$\mathit{nmsort} :: {}'a\ \mathit{list} \Rightarrow {}'a\ \mathit{list}$

$\mathit{nmsort}\ xs = \mathit{merge\_all}\ (\mathit{runs}\ xs)$

2.6.1 Correctness

We have

$$(\forall xs\ ys.\ f\ (xs\ @\ ys) = f\ xs\ @\ ys) \longrightarrow$$
$$\mathit{mset\_mset}\ (\mathit{asc}\ x\ f\ ys) = \{\!\{x\}\!\} + \mathit{mset}\ (f\ []) + \mathit{mset}\ ys \tag{2.21}$$

$$\mathit{mset\_mset}\ (\mathit{desc}\ x\ xs\ ys) = \{\!\{x\}\!\} + \mathit{mset}\ xs + \mathit{mset}\ ys \tag{2.22}$$

$$\mathit{mset\_mset}\ (\mathit{runs}\ xs) = \mathit{mset}\ xs \tag{2.23}$$

$$\mathit{mset}\ (\mathit{nmsort}\ xs) = \mathit{mset}\ xs \tag{2.24}$$

where (2.23), (2.21), and (2.22) are proved simultaneously. The assumption of (2.21) on f ensures that f is a difference list. We use (2.23) together with (2.17) in order to show (2.24). Moreover, we have

$$\forall x \in \mathit{set}\ (\mathit{runs}\ xs).\ \mathit{sorted}\ x \tag{2.25}$$

$$\mathit{sorted}\ (\mathit{nmsort}\ xs) \tag{2.26}$$

where we use (2.25) together with (2.18) to obtain (2.26).

2.6.2 Running Time

Once more, we only count comparisons:

$$
\begin{aligned}
&C_{\mathit{runs}} :: {'a}\ \mathit{list} \Rightarrow \mathit{nat}\\
&C_{\mathit{runs}}\ (a \mathbin{\#} b \mathbin{\#} xs) = 1 + (\textbf{if}\ b < a\ \textbf{then}\ C_{\mathit{desc}}\ b\ xs\ \textbf{else}\ C_{\mathit{asc}}\ b\ xs)\\
&C_{\mathit{runs}}\ [] = 0\\
&C_{\mathit{runs}}\ [\_] = 0\\
\\
&C_{\mathit{asc}} :: {'a} \Rightarrow {'a}\ \mathit{list} \Rightarrow \mathit{nat}\\
&C_{\mathit{asc}}\ a\ (b \mathbin{\#} bs) = 1 + (\textbf{if}\ \neg\ b < a\ \textbf{then}\ C_{\mathit{asc}}\ b\ bs\ \textbf{else}\ C_{\mathit{runs}}\ (b \mathbin{\#} bs))\\
&C_{\mathit{asc}}\ \_\ [] = 0\\
\\
&C_{\mathit{desc}} :: {'a} \Rightarrow {'a}\ \mathit{list} \Rightarrow \mathit{nat}\\
&C_{\mathit{desc}}\ a\ (b \mathbin{\#} bs) = 1 + (\textbf{if}\ b < a\ \textbf{then}\ C_{\mathit{desc}}\ b\ bs\ \textbf{else}\ C_{\mathit{runs}}\ (b \mathbin{\#} bs))\\
&C_{\mathit{desc}}\ \_\ [] = 0\\
\\
&C_{\mathit{nmsort}} :: {'a}\ \mathit{list} \Rightarrow \mathit{nat}\\
&C_{\mathit{nmsort}}\ xs = C_{\mathit{runs}}\ xs + C_{\mathit{merge\_all}}\ (\mathit{runs}\ xs)
\end{aligned}
$$

Again note the mutually recursive definitions of C_{runs}, C_{asc}, and C_{desc}. Hence the remark on proofs about *runs* also applies to proofs about C_{runs}.

Before talking about C_{nmsort}, we need a variant of Lemma 2.5 that also works for lists whose lengths are not powers of two (since the result of *runs* will usually not satisfy this property).

To this end, we will need the following two results, which we prove by two simple computation inductions using (2.15) and (2.13):

$$C_{\mathit{merge\_adj}}\ xss \le |\mathit{concat}\ xss| \tag{2.27}$$

$$|\mathit{concat}\ (\mathit{merge\_adj}\ xss)| = |\mathit{concat}\ xss| \tag{2.28}$$

Lemma 2.7. $C_{\mathit{merge\_all}}\ xss \le |\mathit{concat}\ xss| \cdot \lceil \mathit{lg}\ |xss| \rceil$

Proof by induction on the computation of $C_{\mathit{merge\_all}}$. We concentrate on the nontrivial recursive case arising from the third equation. It follows that xss is of the form $xs\ \#\ ys\ \#\ zss$. Further note that for all $n :: nat$:

$$2 \le n \longrightarrow \lceil \mathit{lg}\ n \rceil = \lceil \mathit{lg}\ ((n - 1)\ \mathrm{div}\ 2 + 1) \rceil + 1 \tag{2.29}$$

Now, let $m = |\mathit{concat}\ xss|$. Then we have

$$\begin{aligned}
& C_{\mathit{merge\_all}}\ xss \\
&= C_{\mathit{merge\_adj}}\ xss + C_{\mathit{merge\_all}}\ (\mathit{merge\_adj}\ xss) \\
&\le m + C_{\mathit{merge\_all}}\ (\mathit{merge\_adj}\ xss) && \text{using (2.27)} \\
&\le m + |\mathit{concat}\ (\mathit{merge\_adj}\ xss)| \cdot \lceil \mathit{lg}\ |\mathit{merge\_adj}\ xss| \rceil && \text{by IH} \\
&= m + m \cdot \lceil \mathit{lg}\ |\mathit{merge\_adj}\ xss| \rceil && \text{by (2.28)} \\
&= m + m \cdot \lceil \mathit{lg}\ ((|xss| + 1)\ \mathrm{div}\ 2) \rceil && \text{by (2.16)} \\
&= m + m \cdot \lceil \mathit{lg}\ ((|zss| + 1)\ \mathrm{div}\ 2 + 1) \rceil \\
&= m \cdot (\lceil \mathit{lg}\ ((|zss| + 1)\ \mathrm{div}\ 2 + 1) \rceil + 1) \\
&= m \cdot \lceil \mathit{lg}\ (|zss| + 2) \rceil && \text{by (2.29)} \\
&= m \cdot \lceil \mathit{lg}\ |xss| \rceil && \square
\end{aligned}$$

Three simple computation inductions, each performed simultaneously for the corresponding mutually recursive definitions, yield:

$$\begin{aligned}
&(\forall xs\ ys.\ f\ (xs\ @\ ys) = f\ xs\ @\ ys) \longrightarrow \\
&|\mathit{concat}\ (\mathit{asc}\ a\ f\ ys)| = 1 + |f\ []| + |ys|, \\
&|\mathit{concat}\ (\mathit{desc}\ a\ xs\ ys)| = 1 + |xs| + |ys|, \\
&|\mathit{concat}\ (\mathit{runs}\ xs)| = |xs|
\end{aligned} \tag{2.30}$$

$$\begin{aligned}
&(\forall xs\ ys.\ f\ (xs\ @\ ys) = f\ xs\ @\ ys) \longrightarrow |\mathit{asc}\ a\ f\ ys| \le 1 + |ys|, \\
&|\mathit{desc}\ a\ xs\ ys| \le 1 + |ys|,\ |\mathit{runs}\ xs| \le |xs|
\end{aligned} \tag{2.31}$$

$$C_{\mathit{asc}}\ a\ ys \le |ys|,\ C_{\mathit{desc}}\ a\ ys \le |ys|,\ C_{\mathit{runs}}\ xs \le |xs| - 1 \tag{2.32}$$

At this point we obtain an upper bound on the number of comparisons required by C_{nmsort}.

Lemma 2.8. $|xs| = n \longrightarrow C_{\mathit{nmsort}}\ xs \le n + n \cdot \lceil \mathit{lg}\ n \rceil$

Proof. Note that

$$C_{\mathit{merge\_all}}\ (\mathit{runs}\ xs) \le n \cdot \lceil \mathit{lg}\ n \rceil \tag{$\star$}$$

as shown by this derivation:

$$\begin{aligned}
&\mathit{C_{merge\_all}}\ (\mathit{runs}\ xs)\\
&\le |\mathit{concat}\ (\mathit{runs}\ xs)| \cdot \lceil \mathit{lg}\ |\mathit{runs}\ xs|\rceil && \text{by Lemma 2.7 with } xss = \mathit{runs}\ xs\\
&\le n \cdot \lceil \mathit{lg}\ |\mathit{runs}\ xs|\rceil && \text{by (2.30)}\\
&\le n \cdot \lceil \mathit{lg}\ n\rceil && \text{by (2.31)}
\end{aligned}$$

We conclude the proof by:

$$\begin{aligned}
&\mathit{C_{nmsort}}\ xs = \mathit{C_{runs}}\ xs + \mathit{C_{merge\_all}}\ (\mathit{runs}\ xs)\\
&\le n + n \cdot \lceil \mathit{lg}\ n\rceil && \text{using (2.32) and } (\star)
\end{aligned}$$

□

2.7 Uniqueness of Sorting

We have seen many different sorting functions now and it may come as a surprise that they are all the same in the sense that they are all *extensionally equal*: they have the same input/output behaviour (but of course not the same running time).

A more abstract formulation of this is that the result of sorting a list is uniquely determined by the specification of sorting. This is what we call the **uniqueness of sorting**: Consider lists whose elements are sorted w.r.t. some linear order. Then any two such lists with the same multiset of elements are equal. Formally:

Theorem 2.9 (Uniqueness of sorting)**.**
$\mathit{mset}\ ys = \mathit{mset}\ xs \wedge \mathit{sorted}\ xs \wedge \mathit{sorted}\ ys \longrightarrow xs = ys$

Proof by induction on xs (for arbitrary ys). The base case is trivial. In the induction step, $xs = x \# xs'$. Thus ys must also be of the form $y \# ys'$ (otherwise their multisets could not be equal).

Thus we now have to prove $x \# xs' = y \# ys'$, and the facts that we have available to do this are

$$\mathit{mset}\ (x \# xs') = \mathit{mset}\ (y \# ys') \tag{2.33}$$
$$\mathit{sorted}\ (x \# xs') \wedge \mathit{sorted}\ (y \# ys') \tag{2.34}$$

and the induction hypothesis

$$\forall ys'.\ \mathit{mset}\ xs' = \mathit{mset}\ ys' \wedge \mathit{sorted}\ xs' \wedge \mathit{sorted}\ ys' \longrightarrow xs' = ys'\ . \tag{IH}$$

Our first objective now is to show that $x = y$. Either $x \le y$ or $x \ge y$ must hold. Let us first prove $x = y$ for the case $x \le y$. From (2.33), we have $x \in_{\#} \mathit{mset}\ (x \# xs') = \mathit{mset}\ (y \# ys')$. Thus x is contained somewhere in the list $y \# ys'$. Since $y \# ys'$ is sorted, all elements of $y \# ys'$ are $\ge y$; in particular we then have $x \ge y$. Together with $x \le y$, we obtain $x = y$ as desired. The case $x \ge y$ is completely analogous.

Now that we know that $x = y$, the rest of the proof is immediate: From (2.33) we obtain $\mathit{mset}\ xs' = \mathit{mset}\ ys'$, and with that and (2.34), the (IH) tells us that $xs' = ys'$ and we are done. □

This theorem directly implies the extensional equality of all sorting functions that we alluded to earlier. That is, any two functions that satisfy the specification from Section 2.1 are extensionally equal.

Corollary 2.10 (All sorting functions are extensionally equal)**.** *If f and g are functions of type $('a :: linorder)\ list \Rightarrow\ 'a\ list$ such that*

$$\forall zs.\ \mathsf{sorted}\ (f\ zs) \wedge \mathsf{mset}\ (f\ zs) = \mathsf{mset}\ zs$$
$$\forall zs.\ \mathsf{sorted}\ (g\ zs) \wedge \mathsf{mset}\ (g\ zs) = \mathsf{mset}\ zs$$

then $\forall zs.\ f\ zs = g\ zs$; or, equivalently: $f = g$

Proof. We use Theorem 2.9 with the instantiations $xs = f\ zs$ and $ys = g\ zs$. □

Note that for both of these theorems, the *linorder* constraint on the element type is crucial: if we have an order $\preceq$ that is *not* linear, then there are elements x, y with $x \preceq y$ and $y \preceq x$ but $x \neq y$. Consequently, the lists $[x,y]$ and $[y,x]$ are not equal, even though they are both sorted w.r.t. $\preceq$ and contain the same elements.

2.8 Stability

A sorting function is called **stable** if the order of equal elements is preserved. However, this only makes a difference if elements are not identified with their keys, as we have done so far. Let us assume instead that sorting is parameterized with a key function $f :: 'a \Rightarrow 'k$ that maps an element to its key and that the keys $'k$ are linearly ordered, not the elements. This is the specification of a sorting function *sort_key*:

$$\mathsf{mset}\ (sort\_key\ f\ xs) = \mathsf{mset}\ xs$$
$$\mathsf{sorted}\ (\mathsf{map}\ f\ (sort\_key\ f\ xs))$$

Assuming (for simplicity) we are sorting pairs of keys and some attached information, stability means that sorting $[(2,\ x),\ (1,\ z),\ (1,\ y)]$ yields $[(1,\ z),\ (1,\ y),\ (2,\ x)]$ and not $[(1,\ y),\ (1,\ z),\ (2,\ x)]$. That is, if we extract all elements with the same key *after* sorting xs, they should be in the same order as in xs:

$$\mathsf{filter}\ (\lambda y.\ f\ y = k)\ (sort\_key\ f\ xs) = \mathsf{filter}\ (\lambda y.\ f\ y = k)\ xs$$

We will now define insertion sort adapted to keys and verify its correctness and stability.

```
insort1_key :: ('a ⇒ 'k) ⇒ 'a ⇒ 'a list ⇒ 'a list
insort1_key _ x [] = [x]
insort1_key f x (y # ys)
= (if f x ≤ f y then x # y # ys else y # insort1_key f x ys)
```

$$\textit{insort\_key} :: ('a \Rightarrow 'k) \Rightarrow 'a\ list \Rightarrow 'a\ list$$

$$\textit{insort\_key}\ \_\ [] = []$$
$$\textit{insort\_key}\ f\ (x\ \#\ xs) = \textit{insort1\_key}\ f\ x\ (\textit{insort\_key}\ f\ xs)$$

The proofs of the functional correctness properties

$$\textit{mset}\ (\textit{insort\_key}\ f\ xs) = \textit{mset}\ xs$$

$$\textit{sorted}\ (\textit{map}\ f\ (\textit{insort\_key}\ f\ xs)) \tag{2.35}$$

are completely analogous to their counterparts for plain *insort*.

The proof of stability uses three auxiliary properties:

$$(\forall x \in \textit{set}\ xs.\ f\ a \le f\ x) \longrightarrow \textit{insort1\_key}\ f\ a\ xs = a\ \#\ xs \tag{2.36}$$

$$\neg\ P\ x \longrightarrow \textit{filter}\ P\ (\textit{insort1\_key}\ f\ x\ xs) = \textit{filter}\ P\ xs \tag{2.37}$$

$$\begin{gathered}\textit{sorted}\ (\textit{map}\ f\ xs) \wedge P\ x \longrightarrow \\ \textit{filter}\ P\ (\textit{insort1\_key}\ f\ x\ xs) = \textit{insort1\_key}\ f\ x\ (\textit{filter}\ P\ xs)\end{gathered} \tag{2.38}$$

The first one is proved by a case analysis on xs. The other two are proved by induction on xs, using (2.36) in the proof of (2.38).

Lemma 2.11 (Stability of *insort_key*).
$\textit{filter}\ (\lambda y.\ f\ y = k)\ (\textit{insort\_key}\ f\ xs) = \textit{filter}\ (\lambda y.\ f\ y = k)\ xs$

Proof by induction on xs. The base case is trivial. In the induction step we consider the list $a\ \#\ xs$ and perform a case analysis. If $f\ a \neq k$ the claim follows by IH using (2.37). Now assume $f\ a = k$:

$$\begin{aligned}
&\textit{filter}\ (\lambda y.\ f\ y = k)\ (\textit{insort\_key}\ f\ (a\ \#\ xs))\\
&= \textit{filter}\ (\lambda y.\ f\ y = k)\ (\textit{insort1\_key}\ f\ a\ (\textit{insort\_key}\ f\ xs))\\
&= \textit{insort1\_key}\ f\ a\ (\textit{filter}\ (\lambda y.\ f\ y = k)\ (\textit{insort\_key}\ f\ xs))\\
&\hspace{12em} \text{using } f\ a = k\text{, (2.38), (2.35)}\\
&= \textit{insort1\_key}\ f\ a\ (\textit{filter}\ (\lambda y.\ f\ y = k)\ xs) \qquad \text{by IH}\\
&= a\ \#\ \textit{filter}\ (\lambda y.\ f\ y = k)\ xs \qquad \text{using } f\ a = k \text{ and (2.36)}\\
&= \textit{filter}\ (\lambda y.\ f\ y = k)\ (a\ \#\ xs) \qquad \text{using } f\ a = k
\end{aligned}$$

□

2.9 Exercises

Exercise 2.1. Show that T_{insort} achieves its optimal value of $2 \cdot n + 1$ for sorted lists, and its worst-case value of $(n + 1) \cdot (n + 2)$ div 2 for the list *rev* $[0..<n]$.

Exercise 2.2. Function *quicksort* appends the lists returned from the recursive calls. This is expensive because the running time of (@) is linear in the length of its first argument. Define a function *quicksort2* :: $'a\ list \Rightarrow 'a\ list \Rightarrow 'a\ list$ that avoids (@) but accumulates the result in its second parameter via (#) only. Prove *quicksort2* $xs\ ys$ = *quicksort* xs @ ys.

Exercise 2.3. There is one obvious optimisation to the version of quicksort that we studied before: instead of partitioning the list into those elements that are smaller than the pivot and those that are at least as big as the pivot, we can use three-way partitioning:

$$\mathit{partition3} :: 'a \Rightarrow 'a\ list \Rightarrow 'a\ list \times 'a\ list \times 'a\ list$$

$$\begin{aligned}&\mathit{partition3}\ x\ xs\\ &= (\mathit{filter}\ (\lambda y.\ y < x)\ xs,\ \mathit{filter}\ (\lambda y.\ y = x)\ xs,\ \mathit{filter}\ (\lambda y.\ y > x)\ xs)\end{aligned}$$

$$\mathit{quicksort3} :: 'a\ list \Rightarrow 'a\ list$$

$$\begin{aligned}&\mathit{quicksort3}\ [] = []\\ &\mathit{quicksort3}\ (x\ \#\ xs)\\ &= (\mathbf{let}\ (ls,\ es,\ gs) = \mathit{partition3}\ x\ xs\\ &\quad\ \mathbf{in}\ \mathit{quicksort3}\ ls\ @\ x\ \#\ es\ @\ \mathit{quicksort3}\ gs)\end{aligned}$$

Prove that this version of quicksort also produces the correct results.

Exercise 2.4. In this exercise, we will examine the worst-case behaviour of Quicksort, which is e.g. achieved if the input list is already sorted. Consider the time function for Quicksort:

$$T_{\mathit{quicksort}} :: 'a\ list \Rightarrow nat$$

$$\begin{aligned}&T_{\mathit{quicksort}}\ [] = 1\\ &T_{\mathit{quicksort}}\ (x\ \#\ xs) = T_{\mathit{quicksort}}\ (\mathit{filter}\ (\lambda y.\ y < x)\ xs)\ +\\ &\qquad\qquad\qquad\qquad\quad T_{\mathit{quicksort}}\ (\mathit{filter}\ (\lambda y.\ y \geq x)\ xs)\ +\\ &\qquad\qquad\qquad\qquad\quad 2 \cdot T_{\mathit{filter}}\ (\lambda\_.\ 1)\ xs + 1\end{aligned}$$

1. Show that Quicksort takes quadratic time on sorted lists by proving

   $$\mathit{sorted}\ xs \longrightarrow T_{\mathit{quicksort}}\ xs = a \cdot |xs|^2 + b \cdot |xs| + c$$

   for suitable values a, b, c.
2. Show that this is the worst-case running time by proving

$$T_{quicksort}\ xs \le a \cdot |xs|^2 + b \cdot |xs| + c$$

for the values of a, b, c you determined in the previous step.

Exercise 2.5. The definition of *msort* is inefficient in that it calls *length*, *take* and *drop* for each list. Instead we can split the list into two halves by traversing it only once and putting its elements alternately on two piles, for example *halve* [2, 3, 4] ([0], [1]) = ([4, 2, 0], [3, 1]). Define *halve* and *msort2*

$$\mathit{msort2} :: {'a}\ list \Rightarrow {'a}\ list$$

$$\mathit{msort2}\ [] = []$$
$$\mathit{msort2}\ [x] = [x]$$
$$\mathit{msort2}\ xs$$
$$= (\textbf{let}\ (ys_1,\ ys_2) = \mathit{halve}\ xs\ ([],\ [])\ \textbf{in}\ \mathit{merge}\ (\mathit{msort2}\ ys_1)\ (\mathit{msort2}\ ys_2))$$

and prove *mset* $($*msort2* $xs)$ = *mset* xs and *sorted* $($*msort2* $xs)$. (Hint for Isabelle users: The definition of *msort2* is tricky because its termination relies on suitable properties of *halve*.)

Exercise 2.6. Define a tail-recursive variant of *merge_adj*

$$\mathit{merge\_adj2} :: {'a}\ list\ list \Rightarrow {'a}\ list\ list \Rightarrow {'a}\ list\ list$$

(with the same complexity as *merge_adj*, in particular no (@)) and define new variants *merge_all2* and *msort_bu2* of *merge_all* and *msort_bu* that utilize *merge_adj2*. Prove functional correctness of *msort_bu2*:

$$\mathit{mset}\ (\mathit{msort\_bu2}\ xs) = \mathit{mset}\ xs \qquad \mathit{sorted}\ (\mathit{msort\_bu2}\ xs)$$

Note that *merge_adj2* [] xss = *merge_adj* xss is not required.

Exercise 2.7. Adapt some of the sorting algorithms other than *insort* to sorting with keys and prove their correctness and stability.

3 Selection

Manuel Eberl

A topic that is somewhat related to that of sorting is **selection**: given a list xs of length n with some linear order defined on its elements and a natural number $k < n$, return the k-th smallest number in the list (starting with $k = 0$ for the minimal element). If xs is sorted, this is exactly the k-th element of the list.

The defining properties of the selection operation are as follows:

$$\begin{aligned} k < |xs| \longrightarrow{} & |\{\!\!\{y \in_\# \textsf{mset}\ xs \mid y < \textsf{select}\ k\ xs\}\!\!\}| \le k \\ & \wedge\ |\{\!\!\{y \in_\# \textsf{mset}\ xs \mid y > \textsf{select}\ k\ xs\}\!\!\}| < |xs| - k \end{aligned} \tag{3.1}$$

In words: $\textsf{select}\ k\ xs$ has the property that at most k elements in the list are strictly smaller than it and at most $n - k$ are strictly bigger.

These properties fully specify the selection operation, as shown by the following theorem:

Theorem 3.1 (Uniqueness of the selection operation)**.**
If $k < |xs|$ *and*

$$\begin{matrix} |\{\!\!\{z \in_\# \textsf{mset}\ xs \mid z < x\}\!\!\}| \le k & |\{\!\!\{z \in_\# \textsf{mset}\ xs \mid z > x\}\!\!\}| < |xs| - k \\ |\{\!\!\{z \in_\# \textsf{mset}\ xs \mid z < y\}\!\!\}| \le k & |\{\!\!\{z \in_\# \textsf{mset}\ xs \mid z > y\}\!\!\}| < |xs| - k \end{matrix} \tag{3.2}$$

then $x = y$ *.*

Proof. Suppose $x \neq y$ and then w.l.o.g. $x < y$. This implies:

$$\{\!\!\{z \in_\# \textsf{mset}\ xs \mid z \le x\}\!\!\} \subseteq_\# \{\!\!\{z \in_\# \textsf{mset}\ xs \mid z < y\}\!\!\} \tag{3.3}$$

From this we can prove the contradiction $|xs| < |xs|$:

$$\begin{aligned} |xs| &= |\{\!\!\{z \in_\# \textsf{mset}\ xs \mid z \le x\}\!\!\}| + |\{\!\!\{z \in_\# \textsf{mset}\ xs \mid z > x\}\!\!\}| \\ &\le |\{\!\!\{z \in_\# \textsf{mset}\ xs \mid z < y\}\!\!\}| + |\{\!\!\{z \in_\# \textsf{mset}\ xs \mid z > x\}\!\!\}| \\ &< k + (|xs| - k) && \text{using (3.2), (3.3)} \\ &= |xs| \end{aligned}$$

□

An equivalent, more concrete definition is the following:

$$\textsf{select} :: nat \Rightarrow {}'a\ list \Rightarrow {}'a$$
$$\textsf{select}\ k\ xs = \textsf{sort}\ xs\ !\ k \tag{3.4}$$

Theorem 3.2. *$\textsf{select}$ as defined by Equation (3.4) satisfies the conditions (3.1).*

Proof. If ys is sorted, a straightforward induction on ys shows the following:

$$\{\!\!\{x \in_\# \textsf{mset}\ ys \mid x < ys\ !\ k\}\!\!\} \subseteq_\# \textsf{mset}\ (\textsf{take}\ k\ ys)$$
$$\{\!\!\{x \in_\# \textsf{mset}\ ys \mid x > ys\ !\ k\}\!\!\} \subseteq_\# \textsf{mset}\ (\textsf{drop}\ (k+1)\ ys)$$

Taking the size of the multisets on both sides, we obtain:

$$|\{\!\!\{x \in_\# \textsf{mset}\ ys \mid x < ys\ !\ k\}\!\!\}| \le k$$
$$|\{\!\!\{x \in_\# \textsf{mset}\ ys \mid x > ys\ !\ k\}\!\!\}| < |ys| - k$$

Now, for an arbitrary list xs, we instantiate the above with $ys := \textsf{sort}\ xs$ and obtain:

$$\begin{aligned} k &\ge |\{\!\!\{x \in_\# \textsf{mset}\ (\textsf{sort}\ xs) \mid x < \textsf{sort}\ xs\ !\ k\}\!\!\}| \\ &= |\{\!\!\{x \in_\# \textsf{mset}\ xs \mid x < \textsf{sort}\ xs\ !\ k\}\!\!\}| && \text{using } \textsf{mset}\ (\textsf{sort}\ xs) = \textsf{mset}\ xs \\ &= |\{\!\!\{x \in_\# \textsf{mset}\ xs \mid x < \textsf{select}\ k\ xs\}\!\!\}| && \text{using (3.4)} \end{aligned}$$

and analogously for the elements greater than $\textsf{select}\ k\ xs$. □

We will frequently need another important fact about $\textsf{sort}$ and $\textsf{select}$, namely that they are invariant under permutation of the input list:

Lemma 3.3. *Let xs and ys be lists with $\textsf{mset}\ xs = \textsf{mset}\ ys$. Then:*

$$\textsf{sort}\ xs = \textsf{sort}\ ys \tag{3.5}$$
$$\textsf{select}\ k\ xs = \textsf{select}\ k\ ys \tag{3.6}$$

Proof. Equation (3.5) follows directly from Theorem 2.9 (the uniqueness of the $\textsf{sort}$ operation), and (3.6) then follows from (3.5) and our definition of $\textsf{select}$. □

The definition of $\textsf{select}$ in terms of $\textsf{sort}\ xs\ !\ k$ already gives us a straightforward $O(n \lg n)$ algorithm for the selection operation: sort the list with one of our $O(n \lg n)$ sorting algorithms and then return the k-th element of the resulting sorted list. It is also fairly easy to come up with an algorithm that has running time $O(kn)$, i.e. that runs in linear time in n for any fixed k (see Exercise 3.3).

In the remainder of this chapter, we will look at a selection algorithm that achieves $O(n)$ running time *uniformly for all $k < n$* [Blum et al. 1973]. Since a selection algorithm must inspect every element at least once (see Exercise 3.4), this running time is asymptotically optimal.

Exercise 3.1. A simple special case of selection is $\mathit{select}\ 0\ xs$, i.e. the minimum. Implement a linear-time function $\mathit{select0}$ such that

$$xs \neq [] \longrightarrow \mathit{select0}\ xs = \mathit{select}\ 0\ xs$$

and prove this. This function should be tail-recursive and traverse the list exactly once. You need not prove the linear running time (it should be obvious).

Exercise 3.2. How can your $\mathit{select0}$ algorithm be modified to obtain an analogous algorithm $\mathit{select1}$ such that

$$|xs| > 1 \longrightarrow \mathit{select1}\ xs = \mathit{select}\ 1\ xs$$

Do not try to prove the correctness yet; it gets somewhat tedious and you will be able to prove it more easily after the next exercise.

Exercise 3.3.

1. Based on the previous two exercises, implement and prove correct an algorithm $\mathit{select\_fixed}$ that fulfills
   $$k < |xs| \longrightarrow \mathit{select\_fixed}\ k\ xs = \mathit{select}\ k\ xs$$
   The algorithm must be tail-recursive with running time $O(kn)$ and traverse the list exactly once.
   Hint: one approach is to first define a function $\mathit{take\_sort}$ that computes $\mathit{take}\ m$ $(\mathit{sort}\ xs)$ in time $O(mn)$.
2. Prove your $\mathit{select1}$ from the previous exercise correct by showing that it is equivalent to $\mathit{select\_fixed}\ 1$.
3. Define a suitable time function for your $\mathit{select\_fixed}$. Prove that this time function is $O(kn)$, i.e. that
   $$T_{\mathit{select\_fixed}}\ k\ xs \leq C_1 \cdot k \cdot |xs| + C_2 \cdot |xs| + C_3 \cdot k + C_4$$
   for all $k < |xs|$ for some constants C_1 to C_4.
   If you have trouble finding the concrete values for these constants, try proving the result with symbolic constants first and observe what conditions need to be fulfilled in order to make the induction step go through.

Exercise 3.4. Show that if xs is a list of integers with no repeated elements, an algorithm computing the result of $\mathit{select}\ k\ xs$ must examine every single element, i.e. for any index $i < |xs|$, the i-th element can be replaced by some other number such that the result changes. Formally:

$$k < |xs| \wedge i < |xs| \wedge \mathit{distinct}\ xs \longrightarrow$$
$$(\exists z.\ \mathit{select}\ k\ (xs[i := z]) \neq \mathit{select}\ k\ xs)$$

Here, the notation $xs[i := z]$ denotes the list xs where the i-th element has been replaced with z (the first list element, as always, having index 0).

Hint: a lemma you might find useful is that $\lambda k.\ \textsf{select}\ k\ xs$ is injective if xs has no repeated elements.

3.1 A Divide-and-Conquer Approach

As a first step in our attempt to derive an efficient algorithm for selection, recall what we did with the function $\textsf{partition3}$ in the threeway quicksort algorithm in Exercise 2.3: we picked some pivot value x from xs and partitioned the input list xs into the sublists ls, es, and gs of the elements smaller, equal, and greater than x, respectively.

If we do the same for $\textsf{select}\ k\ xs$, there are three possible cases:

- If $k < |ls|$, the element we are looking for is located in ls. To be more precise, it is the k-th smallest element of ls, i.e. $\textsf{select}\ k\ ls$.
- If $k < |ls| + |es|$, the element we are looking for is located in es and must therefore be equal to x.
- Otherwise, the element we are looking for must be located in gs. More precisely, it is the k'-th smallest element of gs where $k' = k - |ls| - |es|$.

This gives us a straightforward recursive divide-and-conquer algorithm for selection. To prove this formally, we first prove the following lemma about the behaviour of $\textsf{select}$ applied to a list of the form $xs\ @\ ys$:

Lemma 3.4.

$$\begin{aligned}&k < |xs| + |ys| \longrightarrow (\forall x{\in}\textsf{set}\ xs.\ \forall y{\in}\textsf{set}\ ys.\ x \le y) \longrightarrow\\ &\textsf{select}\ k\ (xs\ @\ ys)\\ &= (\textbf{if}\ k < |xs|\ \textbf{then}\ \textsf{select}\ k\ xs\ \textbf{else}\ \textsf{select}\ (k - |xs|)\ ys)\end{aligned} \tag{3.7}$$

Proof. The assumptions imply that $\textsf{sort}\ xs\ @\ \textsf{sort}\ ys$ is sorted, so that due to the uniqueness of the $\textsf{sort}$ operation, we have:

$$\textsf{sort}\ (xs\ @\ ys) = \textsf{sort}\ xs\ @\ \textsf{sort}\ ys \tag{3.8}$$

Then:

$$\begin{aligned}&\textsf{select}\ k\ (xs\ @\ ys)\\ &= \textsf{sort}\ (xs\ @\ ys)\ !\ k && \text{using (3.4)}\\ &= (\textsf{sort}\ xs\ @\ \textsf{sort}\ ys)\ !\ k && \text{using (3.8)}\\ &= \textbf{if}\ k < |xs|\ \textbf{then}\ \textsf{sort}\ xs\ !\ k\ \textbf{else}\ \textsf{sort}\ ys\ !\ (k - |xs|)\\ &= \textbf{if}\ k < |xs|\ \textbf{then}\ \textsf{select}\ k\ xs\ \textbf{else}\ \textsf{select}\ (k - |xs|)\ ys && \text{using (3.4)}\end{aligned}$$

□

Now the recurrence outlined before is a direct consequence:

Theorem 3.5 (A recurrence for $\textsf{select}$). *Let $k < |xs|$ and x arbitrary. Then:*

$$
\begin{aligned}
\mathsf{select}\ k\ xs = \mathbf{let}\ & (ls,\ es,\ gs) = \mathsf{partition3}\ x\ xs \\
\mathbf{in}\ & \mathbf{if}\ k < |ls|\ \mathbf{then}\ \mathsf{select}\ k\ ls \\
& \quad \mathbf{else\ if}\ k < |ls| + |es|\ \mathbf{then}\ x \\
& \quad \mathbf{else}\ \mathsf{select}\ (k - |ls| - |es|)\ gs
\end{aligned}
$$

Proof. We have $\mathsf{mset}\ xs = \mathsf{mset}\ ls + \mathsf{mset}\ es + \mathsf{mset}\ gs$ and $|xs| = |ls| + |es| + |gs|$. Then:

$$
\begin{aligned}
& \mathsf{select}\ k\ xs \\
& = \mathsf{select}\ k\ (ls\ @\ es\ @\ gs) && \text{using (3.6)} \\
& = \mathbf{if}\ k < |ls|\ \mathbf{then}\ \mathsf{select}\ k\ ls \\
& \quad \mathbf{else\ if}\ k - |ls| < |es|\ \mathbf{then}\ \mathsf{select}\ (k - |ls|)\ es && \text{using (3.7) twice} \\
& \quad \mathbf{else}\ \mathsf{select}\ (k - |ls| - |es|)\ gs
\end{aligned}
$$

Clearly, $k - |ls| < |es| \longleftrightarrow k < |ls| + |es|$ and $\mathsf{select}\ (k - |ls|)\ es = x$ since $\mathsf{select}\ (k - |ls|)\ es \in \mathsf{set}\ es$ and $\mathsf{set}\ es = \{x\}$ by definition. □

Note that this holds for *any* pivot x. Indeed, x need not even be in the list itself. Therefore, the algorithm (which is also known as **Quickselect** [Hoare 1961] due to its similarities with Quicksort) is partially correct no matter what pivot we choose.

However, like with Quicksort, the number of recursive calls (and thereby the running time) depends strongly on the pivot choice:

- If we always choose a pivot that is smaller than any element in the list or bigger than any element in the list, the algorithm does not terminate at all.
- If we choose the smallest element in the list as a pivot every time, only one element is removed from the list in every recursion step so that we get n recursive calls in total. Since we do a linear amount of work in every step, this leads to a running time of $\Theta(n^2)$.
- If we choose pivots from the list at random, the worst-case running time is again $\Theta(n^2)$, but the expected running time can be shown to be $\Theta(n)$, similarly to the situation in Quicksort. Indeed, it can also be shown that it is very unlikely that the running time is "significantly worse than linear" [Karp 1994, Section 2.5].
- If we choose a pivot that cuts the list in half every time (i.e. at most $\frac{n}{2}$ elements are strictly smaller than the pivot and at most $\frac{n}{2}$ are strictly bigger), we get a recursion depth of at most $\lceil \lg n \rceil$ and, by the **master theorem** [Cormen et al. 2009], a running time of $\Theta(n)$ (assuming we can find such a pivot in linear time).

Clearly, the last case is the most desirable one. An element that cuts the list in half is called a **median** (a concept widely used in statistics).

For lists of odd length, there is a unique element in that list that achieves this, whereas for lists of even length there are two such elements (e.g. for the list [1,2,3,4], both 2 and 3 work). In general, a median need also not necessarily be an element of the list itself.

For our purposes, it is useful to pick one of the list elements as a canonical median and refer to it as *the* median of that list. If the list has even length, we use the smaller of the two medians. This leads us to the following formal definition:

$$\mathit{median} :: {}'a\ \mathit{list} \Rightarrow {}'a$$
$$\mathit{median}\ xs = \mathit{select}\ ((|xs| - 1)\ \mathrm{div}\ 2)\ xs$$

Unfortunately, computing the median of a list is no easier than selection (see Exercise 3.5), so it seems that, for now, this does not really help us.

Exercise 3.5. Show that computing $\mathit{select}\ k\ xs$ can be reduced in linear time to computing the median of a list, i.e. give a function $\mathit{reduce\_select\_median}$ that satisfies

$$\begin{aligned}&xs \neq [] \wedge k < |xs| \longrightarrow\\ &\mathit{reduce\_select\_median}\ k\ xs \neq [] \wedge\\ &\mathit{median}\ (\mathit{reduce\_select\_median}\ k\ xs) = \mathit{select}\ k\ xs\end{aligned}$$

with a time function $T_{\mathit{reduce\_select\_median}}$ with an upper bound of the following form:

$$xs \neq [] \wedge k < |xs| \longrightarrow T_{\mathit{reduce\_select\_median}}\ k\ xs \leq C_1 \cdot |xs| + C_2$$

Prove that your function satisfies this property and that its time function has this upper bound.

3.2 The Median of Medians

We have seen that computing a true median in every recursive step is just as hard as the general selection problem, so using the median as a pivot is not going to work. The natural question now is: is there something that is *almost* as good as a median but easier to compute?

This is indeed the case, and this is where the ingenuity of the algorithm lies: instead of computing the median of *all* the list elements, compute the median of only a small fraction of list elements. To be precise, we do the following:

- chop the list into groups of 5 elements each (possibly with one smaller group at the end if n is not a multiple of 5)
- compute the median of each of the $\lceil \frac{n}{5} \rceil$ groups (which can be done in constant time for each group using e.g. insertion sort, since their sizes are bounded by 5)

- compute the median M of these $\lceil \frac{n}{5} \rceil$ elements (which can be done by a recursive call to the selection algorithm)

We call M the **median of medians**. M is not quite as good a pivot as the true median, but it is still fairly decent:

Theorem 3.6 (Pivoting bounds for the median of medians)**.**
Let xs be a list and let $\prec$ be either $<$ or $>$. Let

$$M := \mathsf{median}\ (\mathsf{map}\ \mathsf{median}\ (\mathsf{chop}\ 5\ xs))$$

where the chop *function cuts a list into groups of a given size as described earlier:*

```
chop :: nat ⇒ 'a list ⇒ 'a list list
chop 0 _ = []
chop _ [] = []
chop s xs = take s xs # chop s (drop s xs)
```

Then: $|\{\!\{y \in_\# \mathsf{mset}\ xs \mid y \prec M\}\!\}| \leq \lceil 0.7 \cdot n + 3 \rceil$

Proof. The result of $\mathsf{chop}\ 5\ xs$ is a list of $\lceil n\ /\ 5 \rceil$ chunks, each of size at most 5, i.e. $|\mathsf{chop}\ 5\ xs| = \lceil n\ /\ 5 \rceil$. Let us split these chunks into two groups according to whether their median is $\prec M$ or $\succeq M$:

$$Y_\prec := \{\!\{ys \in_\# \mathsf{mset}\ (\mathsf{chop}\ 5\ xs) \mid \mathsf{median}\ ys \prec M\}\!\}$$
$$Y_\succeq := \{\!\{ys \in_\# \mathsf{mset}\ (\mathsf{chop}\ 5\ xs) \mid \mathsf{median}\ ys \succeq M\}\!\}$$

We clearly have

$$\mathsf{mset}\ xs = (\textstyle\sum_{ys \leftarrow \mathsf{chop}\ 5\ xs} \mathsf{mset}\ ys) \tag{3.9}$$
$$\mathsf{mset}\ (\mathsf{chop}\ 5\ xs) = Y_\prec + Y_\succeq \tag{3.10}$$
$$\lceil n\ /\ 5 \rceil = |Y_\prec| + |Y_\succeq| \tag{3.11}$$

and since M is the median of the medians of the groups, we also know that:

$$|Y_\prec| < \tfrac{1}{2} \cdot \lceil n\ /\ 5 \rceil \tag{3.12}$$

The core idea of the proof is that any group $ys \in_\# Y_\succeq$ can have at most 2 elements that are $\prec M$:

$$\begin{aligned}
&|\{\!\{y \in_\# \mathsf{mset}\ ys \mid y \prec M\}\!\}| \\
&\leq |\{\!\{y \in_\# \mathsf{mset}\ ys \mid y \prec \mathsf{median}\ ys\}\!\}| && \text{because } ys \in_\# Y_\succeq \\
&\leq |ys|\ \mathrm{div}\ 2 && \text{using (3.1)} \\
&\leq 5\ \mathrm{div}\ 2 = 2
\end{aligned}$$

And of course, since each group has size at most 5, any group in $ys \in_\# Y_\prec$ can contribute at most 5 elements. In summary, we have:

$$\begin{aligned}
&\forall\, ys \in_{\#} Y_{\prec}.\ |\{\!\{y \in_{\#} \textsf{\textit{mset}}\ ys \mid y \prec M\}\!\}| \leq 5 \\
&\forall\, ys \in_{\#} Y_{\succeq}.\ |\{\!\{y \in_{\#} \textsf{\textit{mset}}\ ys \mid y \prec M\}\!\}| \leq 2 \qquad (3.13)
\end{aligned}$$

With this, we can begin our estimate of the number of elements $\prec M$:

$$\begin{aligned}
&\{\!\{y \in_{\#} \textsf{\textit{mset}}\ xs \mid y \prec M\}\!\} \\
&= \{\!\{y \in_{\#} (\textstyle\sum_{ys \leftarrow \textsf{\textit{chop}}\ 5\ xs} \textsf{\textit{mset}}\ ys) \mid y \prec M\}\!\} && \text{using (3.9)} \\
&= \textstyle\sum_{ys \leftarrow \textsf{\textit{chop}}\ 5\ xs} \{\!\{y \in_{\#} \textsf{\textit{mset}}\ ys \mid y \prec M\}\!\} \\
&= \textstyle\sum_{ys \in_{\#} (Y_{\prec}\ +\ Y_{\succeq})} \{\!\{y \in_{\#} \textsf{\textit{mset}}\ ys \mid y \prec M\}\!\} && \text{using (3.10)}
\end{aligned}$$

Taking the size of both sides, we have

$$\begin{aligned}
&|\{\!\{y \in_{\#} \textsf{\textit{mset}}\ xs \mid y \prec M\}\!\}| \\
&\leq \textstyle\sum_{ys \in_{\#} (Y_{\prec}\ +\ Y_{\succeq})} |\{\!\{y \in_{\#} \textsf{\textit{mset}}\ ys \mid y \prec M\}\!\}| \\
&= \textstyle\sum_{ys \in_{\#} Y_{\prec}} |\{\!\{y \in_{\#} \textsf{\textit{mset}}\ ys \mid y \prec M\}\!\}| + \\
&\quad \textstyle\sum_{ys \in_{\#} Y_{\succeq}} |\{\!\{y \in_{\#} \textsf{\textit{mset}}\ ys \mid y \prec M\}\!\}| \\
&\leq (\textstyle\sum_{ys \in_{\#} Y_{\prec}} 5) + (\textstyle\sum_{ys \in_{\#} Y_{\succeq}} 2) && \text{using (3.13)} \\
&= 5 \cdot |Y_{\prec}| + 2 \cdot |Y_{\succeq}| \\
&= 2 \cdot (|Y_{\prec}| + |Y_{\succeq}|) + 3 \cdot |Y_{\prec}| \\
&= 2 \cdot \lceil n\ /\ 5 \rceil + 3 \cdot |Y_{\prec}| && \text{using (3.11)} \\
&\leq 2 \cdot \lceil n\ /\ 5 \rceil + \tfrac{3}{2} \cdot \lceil n\ /\ 5 \rceil && \text{using (3.12)} \\
&\leq 3.5 \cdot \lceil n\ /\ 5 \rceil \\
&\leq \lceil 0.7 \cdot n + 3 \rceil
\end{aligned}$$

The delicate arithmetic reasoning about rounding in the end can thankfully be done fully automatically by Isabelle's `linarith` method. □

3.3 Selection in Linear Time

We now have all the ingredients to write down our algorithm: the base cases (i.e. sufficiently short lists) can be handled using the naive approach of performing insertion sort and then returning the k-th element. For bigger lists, we perform the divide-and-conquer approach outlined in Theorem 3.5 using M as a pivot. We have two recursive calls: one on a list with exactly $\lceil 0.2 \cdot n \rceil$ elements to compute M, and one on a list with at most $\lceil 0.7 \cdot n + 3 \rceil$ elements.

We will still need to show later that this actually leads to a linear-time algorithm, but the fact that $0.7 + 0.2 < 1$ is at least encouraging: intuitively, the "work load" is reduced by at least 10 % in every recursive step, so we should reach the base case in a logarithmic number of steps.

The full algorithm looks like this:

```
chop :: nat ⇒ 'a list ⇒ 'a list list
chop 0 _ = []
chop _ [] = []
chop s xs = take s xs # chop s (drop s xs)

slow_select :: nat ⇒ 'a list ⇒ 'a
slow_select k xs = insort xs ! k

slow_median :: 'a list ⇒ 'a
slow_median xs = slow_select ((|xs| − 1) div 2) xs

mom_select :: nat ⇒ 'a list ⇒ 'a
mom_select k xs
= (if |xs| ≤ 20 then slow_select k xs
   else let M = mom_select ((⌈|xs| / 5⌉ − 1) div 2)
                  (map slow_median (chop 5 xs));
            (ls, es, gs) = partition3 M xs
        in if k < |ls| then mom_select k ls
           else if k < |ls| + |es| then M
           else mom_select (k − |ls| − |es|) gs)
```

Correctness and termination are easy to prove:

Theorem 3.7 (Partial Correctness of *mom_select*)**.** *Let xs be a list and $k < |xs|$. Then if mom_select k xs terminates, we have*

$$\textit{mom\_select}\ k\ xs = \textit{select}\ k\ xs\ .$$

Proof. Straightforward computation induction using Theorem 3.5. □

Theorem 3.8 (Termination of *mom_select*)**.** *Let xs be a list and $k < |xs|$. Then mom_select k xs terminates.*

Proof. We use $|xs|$ as a termination measure. We need to show that it decreases in each of the two recursive calls under the precondition $|xs| > 20$. This is easy to see:

- The list in the first recursive call has length $\lceil |xs|\ /\ 5 \rceil$, which is strictly less than $|xs|$ if $|xs| > 1$.
- The length of the list in the second recursive call is at most $|xs| - 1$: by induction hypothesis, the first recursive call terminates, so by Theorem 3.7 we know that $M = \textit{median}\ (\textit{map median}\ (\textit{chop}\ 5\ xs))$ and thus:

$$
\begin{aligned}
M \in \mathit{set}\ (\mathsf{map}\ \mathsf{median}\ (\mathsf{chop}\ 5\ xs))
&= \{\mathsf{median}\ ys \mid ys \in \mathit{set}\ (\mathsf{chop}\ 5\ xs)\} \\
&\subseteq \textstyle\bigcup_{ys \in \mathit{set}\ (\mathsf{chop}\ 5\ xs)}\ \mathit{set}\ ys \\
&= \mathsf{set}\ xs
\end{aligned}
$$

Hence, $M \in \mathsf{set}\ xs$ but $M \notin \mathsf{set}\ ls$ and $M \notin \mathsf{set}\ gs$ by construction. Since $\mathsf{set}\ ls$ and $\mathsf{set}\ gs$ are subsets of $\mathsf{set}\ xs$, this implies that $|ls| < |xs|$ and $|gs| < |xs|$. So in either of the two cases for the second recursive call, the length decreases by at least 1.

Of course, we will later see that it actually decreases by quite a bit more than that, but this very crude estimate is sufficient to show termination.

□

Exercise 3.6. The recursive definition of *mom_select* handles the cases $|xs| \leq 20$ through the naive algorithm using insertion sort. The constant 20 here seems somewhat arbitrary. Find the smallest constant n_0 for which the algorithm still works. Why do you think 20 was chosen?

Note that in practice it may be sensible to choose a much larger cut-off size than 20 and handle shorter lists with a more direct approach that empirically works well for such short lists.

3.4 Time Functions

It remains to show now that this indeed leads to a linear-time algorithm. The time function for our selection algorithm is as follows:

```
$T_{mom\_select}$ :: nat ⇒ 'a list ⇒ nat
$T_{mom\_select}$ k xs
= 1 + $T_{length}$ xs +
   (if |xs| ≤ 20 then $T_{slow\_select}$ k xs
    else let xss = chop 5 xs;
             ms = map slow_median xss;
             idx = (⌈|xs| / 5⌉ − 1) div 2;
             x = mom_select idx ms;
             (ls, es, gs) = partition3 x xs
         in $T_{mom\_select}$ idx ms + $T_{chop}$ 5 xs + $T_{map}$ $T_{slow\_median}$ xss +
            $T_{partition3}$ x xs + $T_{length}$ ls +
            (if k < |ls| then $T_{mom\_select}$ k ls
             else if k < |ls| + |es| then $T_{length}$ es
             else $T_{mom\_select}$ (k − |ls| − |es|) gs + $T_{length}$ es))
```

We can then prove

$$k < |xs| \longrightarrow \mathit{T_{mom\_select}}\ k\ xs \le \mathit{T'_{mom\_select}}\ |xs| \tag{3.14}$$

where the upper bound $\mathit{T'_{mom\_select}}$ is defined as follows:

$$\begin{array}{l}
\mathit{T'_{mom\_select}} :: \mathit{nat} \Rightarrow \mathit{nat} \\
\mathit{T'_{mom\_select}}\ n \\
= (\textbf{if}\ n \le 20\ \textbf{then}\ 483 \\
\quad \textbf{else}\ \mathit{T'_{mom\_select}}\ \lceil 0.2 \cdot n \rceil + \mathit{T'_{mom\_select}}\ \lceil 0.7 \cdot n + 3 \rceil + 19 \cdot n + 54)
\end{array}$$

The time functions of the auxiliary functions used here can be found in Section B.2 in the appendix. The proof is a simple computation induction using Theorem 3.6 and the time bounds for the auxiliary functions from Chapter B in the appendix.

The next section will be dedicated to showing that $\mathit{T'_{mom\_select}} \in O(n)$.

Exercise 3.7. Show that the upper bound $\lceil 0.7 \cdot n + 3 \rceil$ is fairly tight by giving an infinite family $(xs_i)_{i \in \mathbb{N}}$ of lists with increasing lengths for which more than 70 % of the elements are larger than the median of medians (with chopping size 5). In Isabelle terms: define a function $f :: \mathit{nat} \Rightarrow \mathit{nat\ list}$ such that $\forall n.\ |f\ n| < |f\ (n + 1)|$ and

$$\frac{|\{\!\!\{y \in_{\#} \mathit{mset}\ (f\ n) \mid y > \mathit{mom}\ (f\ n)\}\!\!\}|}{|f\ n|} > 0.7$$

where $\mathit{mom}\ xs = \mathit{median}\ (\mathit{map}\ \mathit{median}\ (\mathit{chop}\ 5\ xs))$.

3.5 “Akra–Bazzi Light”

The function $\mathit{T'_{mom\_select}}$ (let us write it as f for now) satisfies the recurrence

$$n > 20 \longrightarrow f\ n = f\ \lceil 0.2 \cdot n \rceil + f\ \lceil 0.7 \cdot n + 3 \rceil + 19 \cdot n + 54 \tag{3.15}$$

Such divide-and-conquer recurrences are beyond the “normal” master theorem, but a generalisation, the *Akra–Bazzi Theorem* [Akra and Bazzi 1998, Eberl 2017b, Leighton 1996], does apply to them. Let us first abstract the situation a bit and consider the recurrence

$$n > 20 \longrightarrow f\ n = f\ \lceil a \cdot n + b \rceil + f\ \lceil c \cdot n + d \rceil + C_1 \cdot n + C_2$$

where $0 < a,\ b < 1$ and $C_1,\ C_2 > 0$. The Akra–Bazzi Theorem then tells us that such a function is $O(n)$ if (and only if) $a + b < 1$. We will prove the relevant direction of this particular case of the theorem now – “Akra–Bazzi Light”, so to say.

Instead of presenting the full theorem statement and its proof right away, let us take a more explorative approach. What we want to prove in the end is that there are real constants $C_3 > 0$ and C_4 such that $f\ n \leq C_3 \cdot n + C_4$ for all n. Suppose we already knew such constants and now wanted to prove that the inequality holds. For the sake of simplicity of the presentation, we assume $b,\ d \geq 0$, but note that these assumptions are unnecessary and the proof still works for negative b and d if we replace b and d with $\mathit{max}\ 0\ b$ and $\mathit{max}\ 0\ d$.

The obvious approach to show this is by induction on n, following the structure of the recurrence above. To do this, we use **strong induction** (i.e. the induction hypothesis holds for all $m < n$)[1] and a case analysis on $n > n_1$ (where n_1 is some constant we will determine later).

The two cases we have to show in the induction are then:

Base case: $\forall n \leq n_1.\ f\ n \leq C_3 \cdot n + C_4$

Step: $\forall n > n_1.\ (\forall m < n.\ f\ m \leq C_3 \cdot m + C_4) \longrightarrow f\ n \leq C_3 \cdot n + C_4$

We can see that in order to even be able to apply the induction hypothesis in the induction step, we need $\lceil a \cdot n + b \rceil < n$. We can make the estimate[2]

$$\lceil a \cdot n + b \rceil \leq a \cdot n + b + 1 \overset{!}{<} n$$

and then solve for n, which gives us $n \overset{!}{>} \frac{b + 1}{1 - a}$. If we do the same for c and d as well, we get the conditions

$$n_1 \geq \frac{b + 1}{1 - a} \qquad \text{and} \qquad n_1 \geq \frac{d + 1}{1 - c} \tag{3.16}$$

However, it will later turn out that these are implied by the other conditions we will have accumulated anyway.

Now that we have ensured that the basic structure of our induction will work out, let us continue with the two cases.

The base cases ($n \leq n_1$) is fairly uninteresting: we can simply choose C_4 to be big enough to satisfy the equality for all $n \leq n_1$, whatever n_1 is.

In the recursive step, unfolding one step of the recurrence and applying the induction hypothesis leaves us with the proof obligation

$$\begin{aligned} &(C_3 \cdot \lceil a \cdot n + b \rceil + C_4) + (C_3 \cdot \lceil c \cdot n + d \rceil + C_4) + C_1 \cdot n + C_2 \\ &\overset{!}{\leq} C_3 \cdot n + C_4\ , \end{aligned}$$

or, equivalently,

$$C_3 \cdot (\lceil a \cdot n + b \rceil + \lceil c \cdot n + d \rceil - n) + C_1 \cdot n + C_2 + C_4 \overset{!}{\leq} 0\ ,$$

[1] In Isabelle, the corresponding rule is called `less_induct`:
$(\forall n.\ (\forall k < n.\ P\ k) \longrightarrow P\ n) \longrightarrow P\ n$ (where $n :: \mathit{nat}$)

[2] The notation $\overset{!}{<}$ stands for "must be less than". It emphasises that this inequality is not a consequence of what we have shown so far, but something that we still need to show, or in this case something that we need to ensure by adding suitable preconditions.

We estimate the left-hand side like this:

$$\begin{aligned}
& C_3 \cdot (\lceil a \cdot n + b\rceil + \lceil c \cdot n + d\rceil - n) + C_1 \cdot n + C_2 + C_4 \\
&\leq C_3 \cdot ((a \cdot n + b + 1) + (c \cdot n + d + 1) - n) + C_1 \cdot n + C_2 + C_4 \\
&= C_3 \cdot (b + d + 2) + C_2 + C_4 - (C_3 \cdot (1 - a - c) - C_1) \cdot n && (*) \\
&\leq C_3 \cdot (b + d + 2) + C_2 + C_4 - (C_3 \cdot (1 - a - c) - C_1) \cdot n_1 && (\dagger) \\
&\overset{!}{\leq} 0
\end{aligned}$$

The step from (∗) to (†) uses the fact that $n > n_1$ and requires the factor $C_3 \cdot (1 - a - c) - C_1$ in front of the n to be positive, i.e. we need to add the assumption

$$C_3 > \frac{C_1}{1 - a - c}\ . \tag{3.17}$$

The term (†) (which we want to be ≤ 0) is now a constant. If we solve that inequality for C_3, we get the following two additional conditions:

$$n_1 > \frac{b + d + 2}{1 - a - c} \quad \text{and} \quad C_3 \geq \frac{C_1 \cdot n_1 + C_2 + C_4}{(1 - a - c) \cdot n_1 - b - d - 2} \tag{3.18}$$

The former of these directly implies our earlier conditions (3.16), so we can safely discard those now.

Now all we have to do is to find a combination of n_1, C_3, and C_4 that satisfies (3.17) and (3.18). This is straightforward:

$$n_1 := \mathit{max}\ n_0 \left(\left\lceil \frac{b + d + 2}{1 - a - c} \right\rceil + 1\right) \qquad C_4 := \mathit{Max}\ \{f\ n \mid n \leq n_1\}$$
$$C_3 := \mathit{max} \left(\frac{C_1}{1 - a - c}\right) \left(\frac{C_1 \cdot n_1 + C_2 + C_4}{(1 - a - c) \cdot n_1 - b - d - 2}\right)$$

And with that, the induction goes through and we get the following theorem:

Theorem 3.9 (Akra Bazzi Light)**.**

$$\begin{aligned}
& a > 0 \wedge c > 0 \wedge a + c < 1 \wedge C_1 \geq 0 \wedge \\
& (\forall n{>}n_0.\ f\ n = f\ \lceil a \cdot n + b\rceil + f\ \lceil c \cdot n + d\rceil + C_1 \cdot n + C_2) \longrightarrow \\
& (\exists C_3\ C_4.\ \forall n.\ f\ n \leq C_3 \cdot n + C_4)
\end{aligned} \tag{3.19}$$

Applying this to our concrete example, we get our final result, namely that median-of-medians selection runs in worst-case linear time, uniformly for all indices k:

Theorem 3.10. *There are constants C_3 and C_4 such that, for any list xs and any natural number $k < |xs|$:*

$$T_{\mathit{mom\_select}}\ k\ xs \leq C_3 \cdot |xs| + C_4$$

Proof. Our "Akra–Bazzi Light" Theorem (3.19) applied to the recurrence (3.15) gives us constants C_3 and C_4 such that, for any natural number n:

$$T'_{mom\_select}\ n \le C_3 \cdot n + C_4 \tag{3.20}$$

Thus we have:

$$\begin{aligned} & T_{mom\_select}\ k\ xs \\ \le\ & T'_{mom\_select}\ |xs| && (3.14) \\ \le\ & C_3 \cdot |xs| + C_4 && (3.20) \end{aligned}$$

□

Exercise 3.8.

1. Suppose that instead of groups of 5, we now chop into groups of size $l \ge 1$. Prove a corresponding generalisation of Theorem 3.6.
2. Examine (on paper only): how does this affect correctness and running time of our selection algorithm? Why do you think $l = 5$ was chosen?

Chapter Notes

In this chapter, we have seen how to find the k-th largest element in a list containing n elements in time $O(n)$, uniformly for all k. Of course, we did not really talk about the constant coefficients that are hidden behind the $O(n)$ and which determine how efficient that algorithm is in practice. Although median-of-medians selection is guaranteed to run in worst-case linear time and therefore asymptotically time-optimal, other approaches with a worse worst-case running time like $O(n \log n)$ or even $O(n^2)$ may perform better in most situations in practice.

One solution to remedy this is to take a hybrid approach: we can use a selection algorithm that performs well in most situations (e.g. the divide-and-conquer approach from Section 3.1 with a fixed or a random pivot) and only resort to the guaranteed-linear-time algorithm if we notice that we are not making much progress. This is the approach taken by Musser's **Introselect** algorithm [Musser 1997].

Part II

Search Trees

4 Binary Trees

Tobias Nipkow

Binary trees are defined as a recursive data type:

$$\textbf{datatype}\ 'a\ tree = \mathit{Leaf} \mid \mathit{Node}\ ('a\ tree)\ 'a\ ('a\ tree)$$

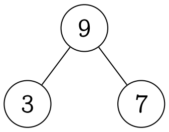
The following syntactic sugar is sprinkled on top:

$$\begin{aligned} \langle\rangle &\equiv \mathit{Leaf} \\ \langle l,\ x,\ r\rangle &\equiv \mathit{Node}\ l\ x\ r \end{aligned}$$

The trees l and r are the left and right **children** of the node $\langle l,\ x,\ r\rangle$.

Because most of our trees will be binary trees, we drop the "binary" most of the time and have also called the type merely *tree*.

When displaying a tree in the usual graphical manner we show only the *Node*s. For example, $\langle\langle\langle\rangle,\ 3,\ \langle\rangle\rangle,\ 9,\ \langle\langle\rangle,\ 7,\ \langle\rangle\rangle\rangle$ is displayed like this:



The (label of the) **root** node is 9. The **depth** (or **level**) of some node (or leaf) in a tree is the distance from the root. The left **spine** of a tree is the sequence of nodes starting from the root and following the left child until that is a leaf. Dually for the right spine. We use these concepts only informally.

4.1 Basic Functions

Two canonical functions on data types are *set* and *map*:

$$\begin{aligned} &\mathit{set\_tree} :: 'a\ tree \Rightarrow 'a\ set \\ &\mathit{set\_tree}\ \langle\rangle = \{\} \\ &\mathit{set\_tree}\ \langle l,\ x,\ r\rangle = \mathit{set\_tree}\ l \cup \{x\} \cup \mathit{set\_tree}\ r \end{aligned}$$

$$\mathsf{map\_tree} :: ('a \Rightarrow 'b) \Rightarrow 'a\ \mathit{tree} \Rightarrow 'b\ \mathit{tree}$$
$$\mathsf{map\_tree}\ f\ \langle\rangle = \langle\rangle$$
$$\mathsf{map\_tree}\ f\ \langle l,\ x,\ r\rangle = \langle \mathsf{map\_tree}\ f\ l,\ f\ x,\ \mathsf{map\_tree}\ f\ r\rangle$$

The *inorder*, *preorder* and *postorder* traversals (we omit the latter) list the elements in a tree in a particular order:

$$\mathsf{inorder} :: 'a\ \mathit{tree} \Rightarrow 'a\ \mathit{list}$$
$$\mathsf{inorder}\ \langle\rangle = []$$
$$\mathsf{inorder}\ \langle l,\ x,\ r\rangle = \mathsf{inorder}\ l\ @\ [x]\ @\ \mathsf{inorder}\ r$$

$$\mathsf{preorder} :: 'a\ \mathit{tree} \Rightarrow 'a\ \mathit{list}$$
$$\mathsf{preorder}\ \langle\rangle = []$$
$$\mathsf{preorder}\ \langle l,\ x,\ r\rangle = x\ \#\ \mathsf{preorder}\ l\ @\ \mathsf{preorder}\ r$$

These two size functions count the number of nodes and leaves in a tree:

$$\mathsf{size} :: 'a\ \mathit{tree} \Rightarrow \mathit{nat}$$
$$|\langle\rangle| = 0$$
$$|\langle l,\ \_,\ r\rangle| = |l| + |r| + 1$$

$$\mathsf{size1} :: 'a\ \mathit{tree} \Rightarrow \mathit{nat}$$
$$|\langle\rangle|_1 = 1$$
$$|\langle l,\ \_,\ r\rangle|_1 = |l|_1 + |r|_1$$

The syntactic sugar $|t|$ for *size* t and $|t|_1$ for *size1* t is only used in this text, not in the Isabelle theories.

Induction proves a convenient fact that explains the name *size1*:

$$|t|_1 = |t| + 1$$

The height (*h*) and the minimal height (*mh*) of a tree are defined as follows:

$h :: {'a}\ tree \Rightarrow nat$

$h\ \langle\rangle = 0$
$h\ \langle l,\ \_,\ r\rangle = max\ (h\ l)\ (h\ r) + 1$

$mh :: {'a}\ tree \Rightarrow nat$

$mh\ \langle\rangle = 0$
$mh\ \langle l,\ \_,\ r\rangle = min\ (mh\ l)\ (mh\ r) + 1$

You can think of them as the longest and shortest (cycle-free) path from the root to a leaf. The names of these functions in the Isabelle theories are *height* and *min_height*. The abbreviations *h* and *mh* are only used in this text.

The obvious properties $h\ t \le |t|$ and $mh\ t \le h\ t$ and the following classical properties have easy inductive proofs:

$$2^{mh\ t} \le |t|_1 \qquad |t|_1 \le 2^{h\ t}$$

We will simply use these fundamental properties without referring to them by a name or number.

The set of subtrees of a tree is defined as follows:

$subtrees :: {'a}\ tree \Rightarrow {'a}\ tree\ set$

$subtrees\ \langle\rangle = \{\langle\rangle\}$
$subtrees\ \langle l,\ a,\ r\rangle = \{\langle l,\ a,\ r\rangle\} \cup subtrees\ l \cup subtrees\ r$

Note that every tree is a subtree of itself.

4.1.1 Exercises

Exercise 4.1. Function *inorder* has quadratic complexity because the running time of (@) is linear in the length of its first argument. Define a function $inorder2 :: {'a}\ tree \Rightarrow {'a}\ list \Rightarrow {'a}\ list$ that avoids (@) but accumulates the result in its second parameter via (#) only. Its running time should be linear in the size of the tree. Prove $inorder2\ t\ xs = inorder\ t\ @\ xs$.

Exercise 4.2. Write a function $enum\_tree :: {'a}\ list \Rightarrow {'a}\ tree\ list$ such that $set\ (enum\_tree\ xs) = \{t \mid inorder\ t = xs\}$ and prove this proposition. You could also prove that *enum_tree* produces lists of *distinct* elements, although that is likely to be harder.

Exercise 4.3. The **weighted path length** of a tree $t :: nat\ tree$ is the sum over all nodes $\langle l,\ w,\ r\rangle$ in t of $w \cdot (d + 1)$ where d is the depth of the node in t:

$$\textsf{wpld} :: \mathit{nat} \Rightarrow \mathit{nat\ tree} \Rightarrow \mathit{nat}$$
$$\textsf{wpld}\ \_\ \langle\rangle = 0$$
$$\textsf{wpld}\ d\ \langle l,\ w,\ r\rangle = (d + 1) \cdot w + \textsf{wpld}\ (d + 1)\ l + \textsf{wpld}\ (d + 1)\ r$$

$$\textsf{wpl0} :: \mathit{nat\ tree} \Rightarrow \mathit{nat}$$
$$\textsf{wpl0}\ t = \textsf{wpld}\ 0\ t$$

The weighted path length can also be defined without the depth parameter:

$$\textsf{wpl} :: \mathit{nat\ tree} \Rightarrow \mathit{nat}$$
$$\textsf{wpl}\ \langle\rangle = 0$$
$$\textsf{wpl}\ \langle l,\ w,\ r\rangle = \textsf{sum\_tree}\ \langle l,\ w,\ r\rangle + \textsf{wpl}\ l + \textsf{wpl}\ r$$

$$\textsf{sum\_tree} :: \mathit{nat\ tree} \Rightarrow \mathit{nat}$$
$$\textsf{sum\_tree}\ \langle\rangle = 0$$
$$\textsf{sum\_tree}\ \langle l,\ n,\ r\rangle = \textsf{sum\_tree}\ l + n + \textsf{sum\_tree}\ r$$

Prove $\textsf{wpl0}\ t = \textsf{wpl}\ t$.

Exercise 4.4. Function *level* lists the elements of a tree on a certain level from left to right:

$$\textsf{level} :: {}'a\ \mathit{tree} \Rightarrow \mathit{nat} \Rightarrow {}'a\ \mathit{list}$$
$$\textsf{level}\ \langle\rangle\ \_ = []$$
$$\textsf{level}\ \langle\_,\ x,\ \_\rangle\ 0 = [x]$$
$$\textsf{level}\ \langle l,\ \_,\ r\rangle\ (n + 1) = \textsf{level}\ l\ n\ @\ \textsf{level}\ r\ n$$

Define a function $\textsf{levels} :: {}'a\ \mathit{tree} \Rightarrow {}'a\ \mathit{list\ list}$ that computes $[\textsf{level}\ t\ 0\ ,\ \ldots,\ \textsf{level}\ t\ (\textsf{h}\ t - 1)]$ (if $t \neq \langle\rangle$ and $\textsf{levels}\ \langle\rangle = []$) but that traverses the tree only once, does not use *nat* but may use auxiliary functions on lists. For starters, prove $|\textsf{levels}\ t| = \textsf{h}\ t$. More challenging is the correctness of *levels* w.r.t. *level*: $n < \textsf{h}\ t \longrightarrow \textsf{levels}\ t\ !\ n = \textsf{level}\ t\ n$

Exercise 4.5. Define a function $\textsf{reconstruct} :: {}'a\ \mathit{list} \Rightarrow {}'a\ \mathit{list} \Rightarrow {}'a\ \mathit{tree}$ that reconstructs a tree from its preorder and inorder traversals. Prove that $\textsf{distinct}\ (\textsf{preorder}\ t) \longrightarrow \textsf{reconstruct}\ (\textsf{preorder}\ t)\ (\textsf{inorder}\ t) = t$.

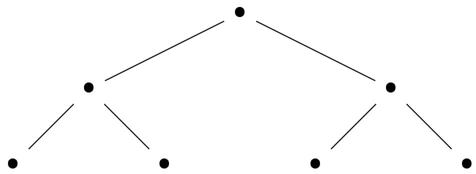


Figure 4.1 A complete tree

Exercise 4.6. Although we focus on binary trees, arbitrarily branching trees can be defined just as easily:

```
datatype 'a rtree = Nd 'a ('a rtree list)
```

Such trees are often called **rose trees**. Define a function $\textsf{mir} :: {}'a\ rtree \Rightarrow {}'a\ rtree$ that mirrors a rose tree and prove $\textsf{mir}\ (\textsf{mir}\ t) = t$.

4.2 Complete Trees

A **complete tree** is one where all the leaves are on the same level. An example is shown in Figure 4.1. The predicate *complete* is defined recursively:

```
complete :: 'a tree ⇒ bool
complete ⟨⟩ = True
complete ⟨l, _, r⟩ = (h l = h r ∧ complete l ∧ complete r)
```

This recursive definition is equivalent with the above definition that all leaves must have the same distance from the root. Formally:

Lemma 4.1. $\textsf{complete}\ t \longleftrightarrow \textsf{mh}\ t = \textsf{h}\ t$

Proof by induction and case analyses on *min* and *max*. □

The following classic property of complete trees is easily proved by induction:

Lemma 4.2. $\textsf{complete}\ t \longrightarrow |t|_1 = 2^{\textsf{h}\ t}$

It turns out below that this is in fact a defining property of complete trees.

For complete trees we have $2^{\textsf{mh}\ t} \le |t|_1 = 2^{\textsf{h}\ t}$. For incomplete trees both $\le$ and $=$ become $<$ as the following two lemmas prove:

Lemma 4.3. $\neg\ \textsf{complete}\ t \longrightarrow |t|_1 < 2^{\textsf{h}\ t}$

Proof by induction. We focus on the induction step where $t = \langle l,\ x,\ r\rangle$. If t is incomplete, there are a number of cases and we prove $|t|_1 < 2^{h\ t}$ in each case. If $h\ l \neq h\ r$, consider the case $h\ l < h\ r$ (the case $h\ r < h\ l$ is symmetric). From $2^{h\ l} < 2^{h\ r}$, $|l|_1 \leq 2^{h\ l}$ and $|r|_1 \leq 2^{h\ r}$ the claim follows: $|t|_1 = |l|_1 + |r|_1 \leq 2^{h\ l} + 2^{h\ r} < 2 \cdot 2^{h\ r} = 2^{h\ t}$. If $h\ l = h\ r$, then either l or r must be incomplete. We consider the case $\neg\ \mathit{complete}\ l$ (the case $\neg\ \mathit{complete}\ r$ is symmetric). From the IH $|l|_1 < 2^{h\ l}$, $|r|_1 \leq 2^{h\ r}$ and $h\ l = h\ r$ the claim follows: $|t|_1 = |l|_1 + |r|_1 < 2^{h\ l} + 2^{h\ r} = 2 \cdot 2^{h\ r} = 2^{h\ t}$. □

Lemma 4.4. $\neg\ \mathit{complete}\ t \longrightarrow 2^{mh\ t} < |t|_1$

The proof of this lemma is completely analogous to the previous proof except that one also needs to use Lemma 4.1.

From the contrapositive of Lemma 4.3 one obtains $|t|_1 = 2^{h\ t} \longrightarrow \mathit{complete}\ t$, the converse of Lemma 4.2. Thus we arrive at:

Corollary 4.5. $\mathit{complete}\ t\ \longleftrightarrow\ |t|_1 = 2^{h\ t}$

The complete trees are precisely the ones where the height is exactly the logarithm of the number of leaves.

4.2.1 Exercises

Exercise 4.7. Define a function *mcs* that computes a maximal complete subtree of some given tree. You are allowed only one traversal of the input but you may freely compute the height of trees and may even compare trees for equality. You are not allowed to use *complete* or *subtrees*.

Prove that *mcs* returns a complete subtree (which should be easy) and that it is maximal in height:

$$u \in \mathit{subtrees}\ t \wedge \mathit{complete}\ u \longrightarrow h\ u \leq h\ (\mathit{mcs}\ t)$$

Bonus: get rid of any tree equality tests in *mcs*.

4.3 Almost Complete Trees

An **almost complete tree** is one where the leaves may occur not just at the lowest level but also one level above:

$$\mathit{acomplete} :: {}'a\ \mathit{tree} \Rightarrow \mathit{bool}$$
$$\mathit{acomplete}\ t = (h\ t - mh\ t \leq 1)$$

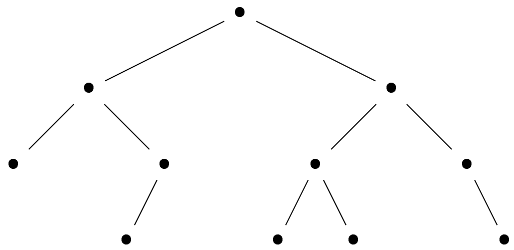


Figure 4.2 An almost complete tree

An example of an almost complete tree is shown in Figure 4.2. You can think of an almost complete tree as a complete tree with (possibly) some additional nodes one level below the last full level.

Almost complete trees are important because among all the trees with the same number of nodes they have minimal height:

Lemma 4.6. $\mathit{acomplete}\ s \wedge |s| \leq |t| \longrightarrow h\ s \leq h\ t$

Proof by cases. If $\mathit{complete}\ s$ then, by Lemma 4.2, $2^{h\ s} = |s|_1 \leq |t|_1 \leq 2^{h\ t}$ and thus $h\ s \leq h\ t$. Now assume $\neg\ \mathit{complete}\ s$. Then Lemma 4.4 yields $2^{mh\ s} < |s|_1 \leq |t|_1 \leq 2^{h\ t}$ and thus $mh\ s < h\ t$. Furthermore we have $h\ s - mh\ s \leq 1$ (from $\mathit{acomplete}\ s$), $h\ s \neq mh\ s$ (from Lemma 4.1) and $mh\ s \leq h\ s$, which together imply $mh\ s + 1 = h\ s$. With $mh\ s < h\ t$ this implies $h\ s \leq h\ t$. □

This is relevant for search trees because their height determines the worst case running time. Almost complete trees are optimal in that sense.

The following lemma yields a closed formula for the height of almost complete trees:

Lemma 4.7. $\mathit{acomplete}\ t \longrightarrow h\ t = \lceil lg\ |t|_1 \rceil$

Proof by cases. If t is complete, the claim follows from Lemma 4.2. Now assume t is incomplete. Then $h\ t = mh\ t + 1$ because $\mathit{acomplete}\ t$, $mh\ t \leq h\ t$ and $\mathit{complete}\ t \longleftrightarrow mh\ t = h\ t$ (Lemma 4.1). Together with $|t|_1 \leq 2^{h\ t}$ this yields $|t|_1 \leq 2^{mh\ t + 1}$ and thus $lg\ |t|_1 \leq mh\ t + 1$. By Lemma 4.4 we obtain $mh\ t < lg\ |t|_1$. These two bounds for $lg\ |t|_1$ together imply the claimed $h\ t = \lceil lg\ |t|_1 \rceil$. □

In the same manner we also obtain:

Lemma 4.8. $\mathit{acomplete}\ t \longrightarrow mh\ t = \lfloor lg\ |t|_1 \rfloor$

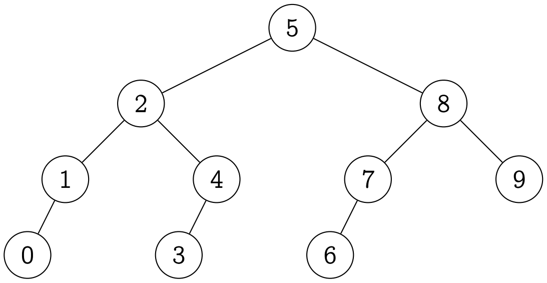


Figure 4.3 Balancing [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

4.3.1 Converting a List into an Almost Complete Tree

We will now see how to convert a list xs into an almost complete tree t such that *inorder* $t = xs$. If the list is sorted, the result is an almost complete binary search tree (see the next chapter). The basic idea is to cut the list in two halves, turn them into almost complete trees recursively and combine them. Cutting up the list in two halves explicitly would lead to an $n \lg n$ algorithm, but we want a linear one. Therefore we use an additional *nat* parameter to tell us how much of the input list should be turned into a tree. The remaining list is returned with the tree:

```
bal :: nat ⇒ 'a list ⇒ 'a tree × 'a list
bal n xs
= (if n = 0 then (⟨⟩, xs)
   else let m = n div 2;
            (l, ys) = bal m xs;
            (r, zs) = bal (n - 1 - m) (tl ys)
        in (⟨l, hd ys, r⟩, zs))
```

The trick is not to chop xs but n in half, because we assume that arithmetic is constant-time. Hence *bal* runs in linear time (see Exercise 4.9). Figure 4.3 shows the result of *bal* 10 [0..9].

Balancing some prefix or all of a list or tree is easily derived:

```
bal_list :: nat ⇒ 'a list ⇒ 'a tree
bal_list n xs = fst (bal n xs)
```

$$\textsf{balance\_list} :: {'a}\ list \Rightarrow {'a}\ tree$$
$$\textsf{balance\_list}\ xs = \textsf{bal\_list}\ |xs|\ xs$$

$$\textsf{bal\_tree} :: nat \Rightarrow {'a}\ tree \Rightarrow {'a}\ tree$$
$$\textsf{bal\_tree}\ n\ t = \textsf{bal\_list}\ n\ (\textsf{inorder}\ t)$$

$$\textsf{balance\_tree} :: {'a}\ tree \Rightarrow {'a}\ tree$$
$$\textsf{balance\_tree}\ t = \textsf{bal\_tree}\ |t|\ t$$

4.3.1.1 Correctness

The following lemma clearly expresses that $\textsf{bal}\ n\ xs$ turns the prefix of length n of xs into a tree and returns the corresponding suffix of xs:

Lemma 4.9. $n \le |xs| \wedge \textsf{bal}\ n\ xs = (t,\ zs) \longrightarrow xs = \textsf{inorder}\ t\ @\ zs \wedge |t| = n$

Proof by complete induction on n, assuming that the proposition holds for all values below n. If $n = 0$ the claim is trivial. Now assume $n \neq 0$ and let $m = n$ div 2 and $m' = n - 1 - m$ (and thus $m,\ m' < n$). From $\textsf{bal}\ n\ xs = (t,\ zs)$ we obtain l, r and ys such that $\textsf{bal}\ m\ xs = (l,\ ys)$, $\textsf{bal}\ m'\ (\textsf{tl}\ ys) = (r,\ zs)$ and $t = \langle l,\ \textsf{hd}\ ys,\ r\rangle$. Because $m < n \le |xs|$ the induction hypothesis implies $xs = \textsf{inorder}\ l\ @\ ys \wedge |l| = m$ $(*)$. This in turn implies $m' \le |\textsf{tl}\ ys|$ and thus the induction hypothesis implies $\textsf{tl}\ ys = \textsf{inorder}\ r\ @\ zs \wedge |r| = m'$ $(**)$. Properties $(*)$ and $(**)$ together with $t = \langle l,\ \textsf{hd}\ ys,\ r\rangle$ imply the claim $xs = \textsf{inorder}\ t\ @\ zs \wedge |t| = n$ because $ys \neq []$. □

The corresponding correctness properties of the derived functions are easy consequences:

$$\begin{array}{rcl} n \le |xs| & \longrightarrow & \textsf{inorder}\ (\textsf{bal\_list}\ n\ xs) = \textsf{take}\ n\ xs \\ & & \textsf{inorder}\ (\textsf{balance\_list}\ xs) = xs \\ n \le |t| & \longrightarrow & \textsf{inorder}\ (\textsf{bal\_tree}\ n\ t) = \textsf{take}\ n\ (\textsf{inorder}\ t) \\ & & \textsf{inorder}\ (\textsf{balance\_tree}\ t) = \textsf{inorder}\ t \end{array}$$

To prove that $\textsf{bal}$ returns an almost complete tree we determine its height and minimal height.

Lemma 4.10. $n \le |xs| \wedge \textsf{bal}\ n\ xs = (t,\ zs) \longrightarrow \textsf{h}\ t = \lceil \textsf{lg}\ (n + 1)\rceil$

Proof. The proof structure is the same as for Lemma 4.9 and we reuse the variable names introduced there. In the induction step we obtain the simplified induction hypothesese $\textsf{h}\ l = \lceil \textsf{lg}\ (m + 1)\rceil$ and $\textsf{h}\ r = \lceil \textsf{lg}\ (m' + 1)\rceil$. This leads to

$$\begin{aligned}
&h\ t = max\ (h\ l)\ (h\ r) + 1\\
&= h\ l + 1 && \text{because } m' \le m\\
&= \lceil lg\ (m + 1) + 1\rceil\\
&= \lceil lg\ (n + 1)\rceil && \text{by (2.29)} \qquad \square
\end{aligned}$$

The following complementary lemma is proved in the same way:

Lemma 4.11. $n \le |xs| \wedge bal\ n\ xs = (t, zs) \longrightarrow mh\ t = \lfloor lg\ (n + 1)\rfloor$

By definition of *acomplete* and because $\lceil x\rceil - \lfloor x\rfloor \le 1$ we obtain that *bal* (and consequently the functions that build on it) returns an almost complete tree:

Corollary 4.12. $n \le |xs| \wedge bal\ n\ xs = (t, ys) \longrightarrow acomplete\ t$

4.3.2 Exercises

Exercise 4.8. Find a formula B such that *acomplete* $\langle l, x, r\rangle = B$ where B may only contain the functions *acomplete*, *complete*, *h*, arithmetic, Boolean operations and l and r. Prove *acomplete* $\langle l, x, r\rangle = B$.

Exercise 4.9. Prove that the running time of function *bal* is linear in its first argument.

4.4 Augmented Trees

A tree of type *'a tree* only stores elements of type *'a*. However, it is frequently necessary to store some additional information of type *'b* in each node too, often for efficiency reasons. Typical examples are:

- The size or the height of the tree. Because recomputing them requires traversing the whole tree.
- Lookup tables where each key of type *'a* is associated with a value of type *'b*.

In this case we simply work with trees of type $('a \times 'b)$ *tree* and call them **augmented trees**. As a result we need to redefine a few functions that should ignore the additional information. For example, function *inorder*, when applied to an augmented tree, should return an *'a list*. Thus we redefine it in the obvious way:

```
inorder :: ('a × 'b) tree ⇒ 'a list

inorder ⟨⟩ = []
inorder ⟨l, (a, _), r⟩ = inorder l @ a # inorder r
```

Another example is *set_tree* :: $('a \times 'b)$ *tree* ⇒ *'a set*. In general, if a function f is originally defined on type *'a tree* but should ignore the *'b*-values in an $('a \times 'b)$ *tree* then we assume that there is a corresponding revised definition of f on augmented trees that focuses on the $'a$-values just like *inorder* above does. Of course functions that do not depend on the information in the nodes, e.g. size and height, stay unchanged.

Note that there are two alternative redefinitions of *inorder* (and similar functions): *map fst* ∘ *inorder* or *inorder* ∘ *map_tree fst* where *inorder* is the original function.

4.4.1 Maintaining Augmented Trees

Maintaining the $'b$-values in an $('a \times 'b)$ *tree* can be hidden inside a suitable smart version of *Node* that has only a constant time overhead. Take the example of augmentation by size:

```
sz :: ('a × nat) tree ⇒ nat
sz ⟨⟩ = 0
sz ⟨_, (_, n), _⟩ = n

node_sz :: ('a × nat) tree ⇒ 'a ⇒ ('a × nat) tree ⇒ ('a × nat) tree
node_sz l a r = ⟨l, (a, sz l + sz r + 1), r⟩
```

A $('a \times nat)$ *tree* satisfies *invar_sz* if the size annotation of every node is computed from its children as specified in *node_sz*:

```
invar_sz :: ('a × nat) tree ⇒ bool
invar_sz ⟨⟩ = True
invar_sz ⟨l, (_, n), r⟩ = (n = sz l + sz r + 1 ∧ invar_sz l ∧ invar_sz r)
```

This predicate is preserved by *node_sz* and guarantees that *sz* returns the size:

$$\textit{invar\_sz}\ l \wedge \textit{invar\_sz}\ r \longrightarrow \textit{invar\_sz}\ (\textit{node\_sz}\ l\ a\ r)$$

$$\textit{invar\_sz}\ t \longrightarrow \textit{sz}\ t = |t|$$

We can generalize this example easily. Assume we have a constant $zero :: 'b$ and a function $f :: 'b \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$ that we iterate over the tree:

```
F :: ('a × 'b) tree ⇒ 'b
F ⟨⟩ = zero
F ⟨l, (a, _), r⟩ = f (F l) a (F r)
```

This generalizes the definition of size. Let *node_f* compute the $'b$-value from the $'b$-values of its children via f:

```
b_val :: ('a × 'b) tree ⇒ 'b
b_val ⟨⟩ = zero
b_val ⟨_, (_, b), _⟩ = b

node_f :: ('a × 'b) tree ⇒ 'a ⇒ ('a × 'b) tree ⇒ ('a × 'b) tree
node_f l a r = ⟨l, (a, f (b_val l) a (b_val r)), r⟩
```

If all $'b$-values are computed as in *node_f*

```
invar_f :: ('a × 'b) tree ⇒ bool
invar_f ⟨⟩ = True
invar_f ⟨l, (a, b), r⟩ = (b = f (b_val l) a (b_val r) ∧ invar_f l ∧ invar_f r)
```

then *b_val* computes F: $\mathit{invar\_f}\ t \longrightarrow \mathit{b\_val}\ t = F\ t$.

4.4.2 Exercises

Exercise 4.10. Augment trees by a pair of a Boolean and something else where the Boolean indicates whether the tree is complete or not. Define *ch*, *node_ch* and *invar_ch* as in Section 4.4.1 and prove the following properties:

$$\mathit{invar\_ch}\ t \longrightarrow \mathit{ch}\ t = (\mathit{complete}\ t,\ ?\ t)$$
$$\mathit{invar\_ch}\ l \wedge \mathit{invar\_ch}\ r \longrightarrow \mathit{invar\_ch}\ (\mathit{node\_ch}\ l\ a\ r)$$

Exercise 4.11. Assume type $'a$ is of class *linorder* and augment each *Node* with the maximum value in that tree. Following Section 4.4.1 (but mind the *option* type!) define $\mathit{mx} :: ('a \times 'b)\ \mathit{tree} \Rightarrow {'b}\ \mathit{option}$, *node_mx* and *invar_mx* and prove

$$\mathit{invar\_mx}\ t \longrightarrow \mathit{mx}\ t = (\textbf{if}\ t = \langle\rangle\ \textbf{then}\ \mathit{None}\ \textbf{else}\ \mathit{Some}\ (\mathit{Max}\ (\mathit{set\_tree}\ t)))$$

where *Max* is the predefined maximum operator on finite, non-empty sets.

5 Binary Search Trees

Tobias Nipkow and Bohua Zhan

The purpose of this chapter is threefold: to introduce **binary search trees** (**BSTs**), to discuss their correctness proofs, and to provide a first example of an abstract data type, a notion discussed in more detail in the next chapter.

Search trees are a means for storing and accessing collections of elements efficiently. In particular they can support sets and maps. We concentrate on sets. We have already seen function *set_tree* that maps a tree to the set of its elements. This is an example of an **abstraction function** that maps concrete data structures to the abstract values that they represent.

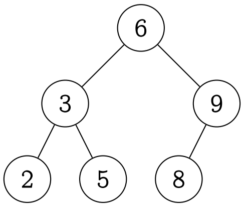
BSTs require a linear ordering on the elements in the tree (as in Chapter 2, Sorting). For each node, the elements in the left child are smaller than the root and the elements in the right child are bigger:

$$\mathit{bst} :: ('a::linorder)\ tree \Rightarrow bool$$

$$\mathit{bst}\ \langle\rangle = \mathit{True}$$

$$\mathit{bst}\ \langle l,\ a,\ r\rangle$$
$$= ((\forall x{\in}\mathit{set\_tree}\ l.\ x < a) \wedge (\forall x{\in}\mathit{set\_tree}\ r.\ a < x) \wedge \mathit{bst}\ l \wedge \mathit{bst}\ r)$$

This is an example of a (coincidentally almost complete) BST:



It is obvious how to search for an element in a BST by comparing the element with the root and descending into one of the two children if you have not found it yet. In the worst case this takes time proportional to the height of the tree. In later chapters we discuss a number of methods for ensuring that the height of the tree is logarithmic in its size. For now we ignore all efficiency considerations and permit our BSTs to degenerate. Thus we call them **unbalanced**.

Exercise 5.1. The above recursive definition of *bst* is not a direct translation of the description “For each node” given in the text. For a more direct translation define a function

$$nodes :: {}'a\ tree \Rightarrow ({}'a\ tree \times {}'a \times {}'a\ tree)\ set$$

that collects all the nodes as triples $(l,\ a,\ r)$. Now define *bst_nodes* as *bst_nodes* $t = (\forall (l,\ a,\ r){\in}nodes\ t.\ ?\ l\ a\ r)$ and prove *bst_nodes* t = *bst* t.

5.1 Interface

Trees are concrete data types that provide the building blocks for implementing abstract data types like sets. The abstract type has a fixed interface, i.e. set of operations, through which the values of the abstract type can be manipulated. The interface hides all implementation detail. In the Search Trees part of the book we focus on the abstract type of sets with the following interface:

```
empty :: 's
insert :: 'a ⇒ 's ⇒ 's
delete :: 'a ⇒ 's ⇒ 's
isin :: 's ⇒ 'a ⇒ bool
```

where $'s$ is the type of sets of elements of type $'a$. Most of our implementations of sets will be based on variants of BSTs and will require a linear order on $'a$, but the general interface does not require this. The correctness of an implementation of this interface will be proved by relating it back to HOL’s type $'a\ set$ via an abstraction function, e.g. *set_tree*.

5.2 Implementing Sets via Unbalanced BSTs

So far we have compared elements via $=$, $\leq$ and $<$. Now we switch to a comparator-based approach:

```
datatype cmp_val = LT | EQ | GT

cmp :: ('a:: linorder) ⇒ 'a ⇒ cmp_val
cmp x y = (if x < y then LT else if x = y then EQ else GT)
```

We will frequently phrase algorithms in terms of *cmp*, *LT*, *EQ* and *GT* instead of $<$, $=$ and $>$. This leads to more symmetric code. If some type comes with its own primitive *cmp* function this can yield a speed-up over the above generic *cmp* function.

Below you find an implementation of the set interface in terms of BSTs. Functions *isin* and *insert* are self-explanatory. Deletion is more interesting.

```
empty :: 'a tree
empty = ⟨⟩

isin :: 'a tree ⇒ 'a ⇒ bool
isin ⟨⟩ _ = False
isin ⟨l, a, r⟩ x
= (case cmp x a of LT ⇒ isin l x | EQ ⇒ True | GT ⇒ isin r x)

insert :: 'a ⇒ 'a tree ⇒ 'a tree
insert x ⟨⟩ = ⟨⟨⟩, x, ⟨⟩⟩
insert x ⟨l, a, r⟩ = (case cmp x a of
                      LT ⇒ ⟨insert x l, a, r⟩ |
                      EQ ⇒ ⟨l, a, r⟩ |
                      GT ⇒ ⟨l, a, insert x r⟩)

delete :: 'a ⇒ 'a tree ⇒ 'a tree
delete _ ⟨⟩ = ⟨⟩
delete x ⟨l, a, r⟩
= (case cmp x a of
    LT ⇒ ⟨delete x l, a, r⟩ |
    EQ ⇒ if r = ⟨⟩ then l else let (a', r') = split_min r in ⟨l, a', r'⟩ |
    GT ⇒ ⟨l, a, delete x r⟩)

split_min :: 'a tree ⇒ 'a × 'a tree
split_min ⟨l, a, r⟩
= (if l = ⟨⟩ then (a, r) else let (x, l') = split_min l in (x, ⟨l', a, r⟩))
```

5.2.1 Deletion

Function *delete* deletes a from $\langle l,\ a,\ r\rangle$ (where $r \neq \langle\rangle$) by replacing a with a' and r with r' where

a' is the leftmost (least) element of r, also called the inorder successor of a,

r' is the remainder of r after removing a'.

We call this **deletion by replacing**. Of course one can also obtain a' as the inorder predecessor of a in l.

An alternative is to delete a from $\langle l,\ a,\ r\rangle$ by “joining” l and r:

```
delete2 :: 'a ⇒ 'a tree ⇒ 'a tree
delete2 _ ⟨⟩ = ⟨⟩
delete2 x ⟨l, a, r⟩ = (case cmp x a of
                         LT ⇒ ⟨delete2 x l, a, r⟩ |
                         EQ ⇒ join l r |
                         GT ⇒ ⟨l, a, delete2 x r⟩)

join :: 'a tree ⇒ 'a tree ⇒ 'a tree
join t ⟨⟩ = t
join ⟨⟩ t = t
join ⟨t1, a, t2⟩ ⟨t3, b, t4⟩
= (case join t2 t3 of
    ⟨⟩ ⇒ ⟨t1, a, ⟨⟨⟩, b, t4⟩⟩ |
    ⟨u2, x, u3⟩ ⇒ ⟨⟨t1, a, u2⟩, x, ⟨u3, b, t4⟩⟩)
```

We call this **deletion by joining**. The characteristic property of *join* is that $inorder\ (join\ l\ r) = inorder\ l\ @\ inorder\ r$.

The definition of *join* may appear needlessly complicated. Why not this much simpler version:

```
join0 t ⟨⟩ = t
join0 ⟨⟩ t = t
join0 ⟨t1, a, t2⟩ ⟨t3, b, t4⟩ = ⟨t1, a, ⟨join0 t2 t3, b, t4⟩⟩
```

Because, with this version of *join*, deletion may almost double the height of the tree, in contrast to *join* and also deletion by replacing, where the height cannot increase:

Exercise 5.2. First prove that *join* behaves well:

$$h\ (join\ l\ r) \le max\ (h\ l)\ (h\ r) + 1$$

Now show that *join0* behaves badly: find an upper bound ub of $h\ (join0\ l\ r)$ such that ub is a function of $h\ l$ and $h\ r$. Prove $h\ (join0\ l\ r) \le ub$ and prove that ub is a tight upper bound if l and r are complete trees.

We focus on *delete*, deletion by replacing, in the rest of the chapter.

5.3 Correctness

Why is the above implementation correct? Roughly speaking, because the implementations of *empty*, *insert*, *delete* and *isin* on type *'a tree* simulate the behaviour of

$\{\}$, $\cup$, $-$ and $\in$ on type $'a\ set$. Taking the abstraction function into account we can formulate the simulation precisely:

$$\mathit{set\_tree}\ \mathit{empty} = \{\}$$
$$\mathit{set\_tree}\ (\mathit{insert}\ x\ t) = \mathit{set\_tree}\ t \cup \{x\}$$
$$\mathit{set\_tree}\ (\mathit{delete}\ x\ t) = \mathit{set\_tree}\ t - \{x\}$$
$$\mathit{isin}\ t\ x = (x \in \mathit{set\_tree}\ t)$$

However, the implementation only works correctly on BSTs. Therefore we need to add the precondition *bst* t to all but the first proposition. Why are we permitted to assume this precondition? Only because *bst* is an **invariant** of this implementation: *bst* holds for *empty*, and both *insert* and *delete* preserve *bst*. Therefore every tree that can be manufactured through the interface is a BST. Of course this adds another set of proof obligations for correctness, **invariant preservation**:

$$\mathit{bst}\ \mathit{empty}$$
$$\mathit{bst}\ t \longrightarrow \mathit{bst}\ (\mathit{insert}\ x\ t)$$
$$\mathit{bst}\ t \longrightarrow \mathit{bst}\ (\mathit{delete}\ x\ t)$$

When looking at the abstract data type of sets from the user (or “client”) perspective, we would call the collection of all proof obligations for the correctness of an implementation the **specification** of the abstract type.

Exercise 5.3. Verify the implementation in Section 5.2 by showing all the proof obligations above, without the detour via sorted lists explained below.

Exercise 5.4. Define a function *union_tree* $::$ $('a::linorder)\ tree \Rightarrow\ 'a\ tree \Rightarrow\ 'a\ tree$ and prove *set_tree* $(\mathit{union\_tree}\ t_1\ t_2) = \mathit{set\_tree}\ t_1 \cup \mathit{set\_tree}\ t_2$ and *bst* $(\mathit{union\_tree}\ t_1\ t_2)$, assuming *bst* t_1 and *bst* t_2. Hint: define and use an auxiliary function *split_tree* $::$ $('a::linorder) \Rightarrow\ 'a\ tree \Rightarrow\ 'a\ tree \times\ 'a\ tree$ such that *split_tree* $x\ t = (lx,\ gx)$ implies that lx/gx contains those elements in t that are less/greater x.

5.4 Correctness Proofs

It turns out that direct proofs of the properties in the previous section can be cumbersome, at least for *delete*. Yet the correctness of the implementation is quite obvious to most (functional) programmers. Which is why most algorithm texts do not spend any time on functional correctness of search trees and concentrate on non-obvious structural properties that imply the logarithmic height of the trees — of course our simple BSTs do not guarantee the latter.

We will now present how the vague notion of “obvious” can be concretized and automated to such a degree that we do not need to discuss functional correctness of search tree implementations again in this book. This is because our approach is quite generic: it works not only for the BSTs in this chapter but also for the more efficient variants discussed in later chapters. The remainder of this section can be skipped if one is not interested in proof automation.

5.4.1 The Idea

The key idea [Nipkow 2016] is to express *bst* and *set_tree* via *inorder*:

$$\mathit{bst}\ t = \mathit{sorted}\ (\mathit{inorder}\ t) \quad \text{and} \quad \mathit{set\_tree}\ t = \mathit{set}\ (\mathit{inorder}\ t)$$

where

```
sorted :: 'a list ⇒ bool
sorted [] = True
sorted [_] = True
sorted (x # y # zs) = (x < y ∧ sorted (y # zs))
```

Note that this is "sorted w.r.t. $(<)$" whereas in the chapter on sorting *sorted* was defined as "sorted w.r.t. $(\leq)$".

Instead of showing directly that BSTs implement sets, we show that they implement an intermediate specification based on lists (and later that the list-based specification implies the set-based one). We can assume that the lists are *sorted* because they are abstractions of BSTs. Insertion and deletion on sorted lists can be defined as follows:

```
ins_list :: 'a ⇒ 'a list ⇒ 'a list
ins_list x [] = [x]
ins_list x (a # xs)
= (if x < a then x # a # xs
   else if x = a then a # xs else a # ins_list x xs)

del_list :: 'a ⇒ 'a list ⇒ 'a list
del_list _ [] = []
del_list x (a # xs) = (if x = a then xs else a # del_list x xs)
```

The abstraction function from trees to lists is function *inorder*. The specification in Figure 5.1 expresses that *empty*, *insert*, *delete* and *isin* implement [], *ins_list*, *del_list* and $\lambda xs\ x.\ x \in \mathit{set}\ xs$. One nice aspect of this specification is that it does not require us to prove invariant preservation explicitly: it follows from the fact (proved below) that *ins_list* and *del_list* preserve *sorted*.

$$\textit{inorder empty} = []$$
$$\textit{sorted}\ (\textit{inorder}\ t) \longrightarrow \textit{inorder}\ (\textit{insert}\ x\ t) = \textit{ins\_list}\ x\ (\textit{inorder}\ t)$$
$$\textit{sorted}\ (\textit{inorder}\ t) \longrightarrow \textit{inorder}\ (\textit{delete}\ x\ t) = \textit{del\_list}\ x\ (\textit{inorder}\ t)$$
$$\textit{sorted}\ (\textit{inorder}\ t) \longrightarrow \textit{isin}\ t\ x = (x \in \textit{set}\ (\textit{inorder}\ t))$$

Figure 5.1 List-based Specification of BSTs

5.4.2 BSTs Implement Sorted Lists — A Framework

We present a library of lemmas that automate the functional correctness proofs for the BSTs in this chapter and the more efficient variants in later chapters. This library is motivated by general considerations concerning the shape of formulas that arise during verification.

As a motivating example we examine how to prove

$$\textit{sorted}\ (\textit{inorder}\ t) \longrightarrow \textit{inorder}\ (\textit{insert}\ x\ t) = \textit{ins\_list}\ x\ (\textit{inorder}\ t)$$

The proof is by induction on t and we consider the case $t = \langle l, a, r\rangle$ such that $x < a$. Ideally the proof looks like this:

$$\begin{aligned}&\textit{inorder}\ (\textit{insert}\ x\ t) = \textit{inorder}\ (\textit{insert}\ x\ l)\ @\ a\ \#\ \textit{inorder}\ r\\ &= \textit{ins\_list}\ x\ (\textit{inorder}\ l)\ @\ a\ \#\ \textit{inorder}\ r\\ &= \textit{ins\_list}\ x\ (\textit{inorder}\ l\ @\ a\ \#\ \textit{inorder}\ r) = \textit{ins\_list}\ x\ t\end{aligned}$$

The first and last step are by definition, the second step by induction hypothesis, and the third step by lemmas in Figure 5.2: (5.1) rewrites the assumption *sorted* $(\textit{inorder}\ t)$ to *sorted* $(\textit{inorder}\ l\ @\ [a]) \wedge \textit{sorted}\ (a\ \#\ \textit{inorder}\ r)$, thus allowing (5.5) to rewrite $\textit{ins\_list}\ x\ (\textit{inorder}\ l\ @\ a\ \#\ \textit{inorder}\ r)$ to $\textit{ins\_list}\ x\ (\textit{inorder}\ l)\ @\ a\ \#\ \textit{inorder}\ r$.

The lemma library in Figure 5.2 helps to prove the properties in Figure 5.1. These proofs are by induction on t and lead to (possibly nested) tree constructor terms like $\langle\langle t_1, a_1, t_2\rangle, a_2, t_3\rangle$ where the t_i and a_i are variables. Evaluating *inorder* of such a tree leads to a list of the following form:

$$\textit{inorder}\ t_1\ @\ a_1\ \#\ \textit{inorder}\ t_2\ @\ a_2\ \#\ \ldots\ \#\ \textit{inorder}\ t_n$$

Now we discuss the lemmas in Figure 5.2 that simplify the application of *sorted*, *ins_list* and *del_list* to such terms.

Terms of the form $\textit{sorted}\ (xs_1\ @\ a_1\ \#\ xs_2\ @\ a_2\ \#\ \ldots\ \#\ xs_n)$ are decomposed into the following *basic* formulas

$$\mathit{sorted}\ (xs\ @\ y\ \#\ ys) = (\mathit{sorted}\ (xs\ @\ [y]) \wedge \mathit{sorted}\ (y\ \#\ ys)) \tag{5.1}$$

$$\begin{aligned}&\mathit{sorted}\ (x\ \#\ xs\ @\ y\ \#\ ys)\\ &= (\mathit{sorted}\ (x\ \#\ xs) \wedge x < y \wedge \mathit{sorted}\ (xs\ @\ [y]) \wedge \mathit{sorted}\ (y\ \#\ ys))\end{aligned} \tag{5.2}$$

$$\mathit{sorted}\ (x\ \#\ xs) \longrightarrow \mathit{sorted}\ xs \tag{5.3}$$

$$\mathit{sorted}\ (xs\ @\ [y]) \longrightarrow \mathit{sorted}\ xs \tag{5.4}$$

$$\begin{aligned}&\mathit{sorted}\ (xs\ @\ [a]) \Longrightarrow \mathit{ins\_list}\ x\ (xs\ @\ a\ \#\ ys) =\\ &\quad (\textbf{if}\ x < a\ \textbf{then}\ \mathit{ins\_list}\ x\ xs\ @\ a\ \#\ ys\ \textbf{else}\ xs\ @\ \mathit{ins\_list}\ x\ (a\ \#\ ys))\end{aligned} \tag{5.5}$$

$$\begin{aligned}&\mathit{sorted}\ (xs\ @\ a\ \#\ ys) \Longrightarrow \mathit{del\_list}\ x\ (xs\ @\ a\ \#\ ys) =\\ &\quad (\textbf{if}\ x < a\ \textbf{then}\ \mathit{del\_list}\ x\ xs\ @\ a\ \#\ ys\ \textbf{else}\ xs\ @\ \mathit{del\_list}\ x\ (a\ \#\ ys))\end{aligned} \tag{5.6}$$

$$\mathit{sorted}\ (x\ \#\ xs) = ((\forall y{\in}\mathit{set}\ xs.\ x < y) \wedge \mathit{sorted}\ xs) \tag{5.7}$$

$$\mathit{sorted}\ (xs\ @\ [x]) = (\mathit{sorted}\ xs \wedge (\forall y{\in}\mathit{set}\ xs.\ y < x)) \tag{5.8}$$

Figure 5.2 Lemmas for *sorted*, *ins_list*, *del_list*

$$\begin{array}{ll}\mathit{sorted}\ (xs\ @\ [a]) & \text{(simulating } \forall x{\in}\mathit{set}\ xs.\ x < a\text{)}\\ \mathit{sorted}\ (a\ \#\ xs) & \text{(simulating } \forall x{\in}\mathit{set}\ xs.\ a < x\text{)}\\ a < b & \end{array}$$

by the rewrite rules (5.1)–(5.2). Lemmas (5.3)–(5.4) enable deductions from basic formulas.

Terms of the form $\mathit{ins\_list}\ x\ (xs_1\ @\ a_1\ \#\ xs_2\ @\ a_2\ \#\ \ldots\ \#\ xs_n)$ are rewritten with (5.5) (and the defining equations for *ins_list*) to push *ins_list* inwards. Terms of the form $\mathit{del\_list}\ x\ (xs_1\ @\ a_1\ \#\ xs_2\ @\ a_2\ \#\ \ldots\ \#\ xs_n)$ are rewritten with (5.6) (and the defining equations for *del_list*) to push *del_list* inwards. The *isin* property in Figure 5.1 can be proved with the help of (5.1), (5.7) and (5.8).

The lemmas in Figure 5.2 form the complete set of basic lemmas on which the automatic proofs of almost all search trees in the book rest; only splay trees (see Chapter 21) need additional lemmas.

5.4.3 Sorted Lists Implement Sets

It remains to be shown that the list-based specification (Figure 5.1) implies the set-based correctness properties in Section 5.3. Because $\mathit{bst}\ t = \mathit{sorted}\ (\mathit{inorder}\ t)$, the latter correctness properties become

$$\begin{aligned}&\mathit{set\_tree}\ \mathit{empty} = \{\}\\ &\mathit{sorted}\ (\mathit{inorder}\ t) \longrightarrow \mathit{set\_tree}\ (\mathit{insert}\ x\ t) = \mathit{set\_tree}\ t \cup \{x\}\\ &\mathit{sorted}\ (\mathit{inorder}\ t) \longrightarrow \mathit{set\_tree}\ (\mathit{delete}\ x\ t) = \mathit{set\_tree}\ t - \{x\}\end{aligned}$$

$$\mathit{sorted}\ (\mathit{inorder}\ t) \longrightarrow \mathit{isin}\ t\ x = (x \in \mathit{set\_tree}\ t)$$
$$\mathit{sorted}\ (\mathit{inorder}\ \mathit{empty})$$
$$\mathit{sorted}\ (\mathit{inorder}\ t) \longrightarrow \mathit{sorted}\ (\mathit{inorder}\ (\mathit{insert}\ x\ t))$$
$$\mathit{sorted}\ (\mathit{inorder}\ t) \longrightarrow \mathit{sorted}\ (\mathit{inorder}\ (\mathit{delete}\ x\ t))$$

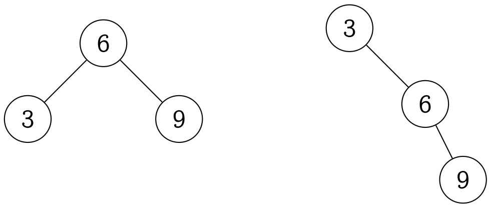
They are proved directly by composing the list-based specification (Figure 5.1, proved above) with the correctness of the sorted list implementation of sets

$$\mathit{set}\ (\mathit{ins\_list}\ x\ xs) = \mathit{set}\ xs \cup \{x\}$$
$$\mathit{sorted}\ xs \longrightarrow \mathit{set}\ (\mathit{del\_list}\ x\ xs) = \mathit{set}\ xs - \{x\}$$
$$\mathit{sorted}\ xs \longrightarrow \mathit{sorted}\ (\mathit{ins\_list}\ x\ xs)$$
$$\mathit{sorted}\ xs \longrightarrow \mathit{sorted}\ (\mathit{del\_list}\ x\ xs)$$

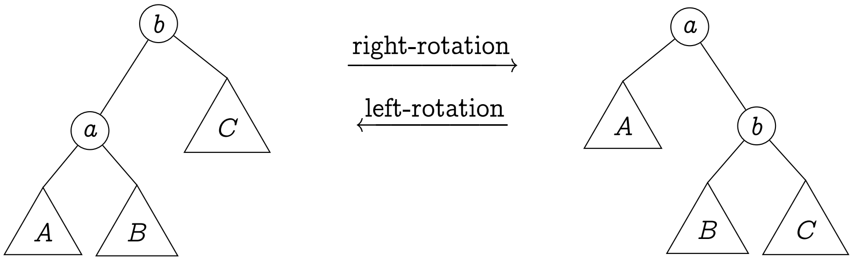
(which have easy inductive proofs) using $\mathit{set\_tree}\ t = \mathit{set}\ (\mathit{inorder}\ t)$.

5.5 Tree Rotations

As discussed in the introduction to this chapter, the BST on the left is better than the one on the right, which has degenerated to a list:



On average, searching for a random key is faster in the left than in the right BST, assuming that all keys are equally likely. In later chapters, a number of balancing schemas will be presented that guarantee logarithmic height (in the number of nodes) of trees balanced according to those schemas. The basic balancing mechanisms are **rotations**, local tree transformations that preserve *inorder* but modify the shape:



We will now show that any two trees t_1 and t_2 with the same *inorder* can be transformed into each other by a linear number of rotations. The basic idea is simple.

Transform t_1 into a list-like tree l by right-rotations. In order to transform l into t_2, note that we can transform t_2 into l (because $\mathit{inorder}\ t_1 = \mathit{inorder}\ t_2$). Hence we merely need to reverse the transformation of t_2 into l.

We call a tree in **list-form** if it is of the form

$$\langle\langle\rangle,\ a_1,\ \langle\langle\rangle,\ a_2,\ \ldots\ \langle\langle\rangle,\ a_n,\ \langle\rangle\rangle\ldots\rangle\rangle$$

Formally:

$\mathit{is\_list} :: {'a}\ \mathit{tree} \Rightarrow \mathit{bool}$

$\mathit{is\_list}\ \langle l,\ \_,\ r\rangle = (l = \langle\rangle \wedge \mathit{is\_list}\ r)$
$\mathit{is\_list}\ \langle\rangle = \mathit{True}$

A tree is in list-form iff no right-rotation is applicable anywhere in the tree. The following function performs right-rotations in a top-down manner along the right spine of a tree:

$\mathit{list\_of} :: {'a}\ \mathit{tree} \Rightarrow {'a}\ \mathit{tree}$

$\mathit{list\_of}\ \langle\langle A,\ a,\ B\rangle,\ b,\ C\rangle = \mathit{list\_of}\ \langle A,\ a,\ \langle B,\ b,\ C\rangle\rangle$
$\mathit{list\_of}\ \langle\langle\rangle,\ a,\ A\rangle = \langle\langle\rangle,\ a,\ \mathit{list\_of}\ A\rangle$
$\mathit{list\_of}\ \langle\rangle = \langle\rangle$

The termination of this function may not be obvious. The problem is the first equation because the size of $\langle\langle A,\ a,\ B\rangle,\ b,\ C\rangle$ and $\langle A,\ a,\ \langle B,\ b,\ C\rangle\rangle$ are the same. However, the right spine has become one longer, which must end when all nodes of the tree are on the right spine. This suggests the measure function $\lambda t.\ |t| - \mathit{rlen}\ t$ where

$\mathit{rlen} :: {'a}\ \mathit{tree} \Rightarrow \mathit{nat}$

$\mathit{rlen}\ \langle\rangle = 0$
$\mathit{rlen}\ \langle\_,\ \_,\ r\rangle = \mathit{rlen}\ r + 1$

This works for the first $\mathit{list\_of}$ equation but not for the second one: $|\langle\langle\rangle,\ a,\ A\rangle| - \mathit{rlen}\ \langle\langle\rangle,\ a,\ A\rangle = |A| - \mathit{rlen}\ A$. Luckily the measure function $\lambda t.\ 2 \cdot |t| - \mathit{rlen}\ t$ decreases with every recursive call, thus proving termination.

The correctness of $\mathit{list\_of}$ is easily expressed

$\mathit{is\_list}\ (\mathit{list\_of}\ t)$

$\mathit{inorder}\ (\mathit{list\_of}\ t) = \mathit{inorder}\ t$

and proved by computation induction.

The claim that only a linear number of rotations is needed cannot be proved from function *list_of* because it does not count the rotations (but see Exercise 5.5). More problematic is the fact that we cannot formalize the second step of our overall proof, namely the idea of reversing the sequence of rotations that *list_of* performs because the rotations are hidden inside *list_of*. Thus we abandon this formalization and restart by introducing an explicit notion of **position** (type *pos*) in a tree:

```
datatype dir = L | R
type_synonym pos = dir list
```

The position of a node in a tree is a sequence of left/right *dir*ections. They encode how to reach that node from the root by turning left or right at each successive node. For example, the position of $\langle\langle\rangle, 1, \langle\rangle\rangle$ in $\langle\langle\langle\rangle, 0, \langle\langle\rangle, 1, \langle\rangle\rangle\rangle, 2, \langle\langle\rangle, 3, \langle\rangle\rangle\rangle$ is [*L*, *R*].

Function *rotR_poss* is the analogue of *list_of* but whereas *list_of* returns the rotated tree, *rotR_poss* produces the list of positions where the rotations should be applied:

```
rotR_poss :: 'a tree ⇒ pos list
rotR_poss ⟨⟨A, a, B⟩, b, C⟩ = [] # rotR_poss ⟨A, a, ⟨B, b, C⟩⟩
rotR_poss ⟨⟨⟩, _, A⟩ = map ((#) R) (rotR_poss A)
rotR_poss ⟨⟩ = []
```

Termination is again proved with the help of the measure function $\lambda t.\ 2 \cdot |t| - \mathit{rlen}\ t$.

Functions *apply_at* and *apply_ats* perform a transformation at a (list of) position(s):

```
apply_at :: ('a tree ⇒ 'a tree) ⇒ pos ⇒ 'a tree ⇒ 'a tree
apply_at f [] t = f t
apply_at f (L # ds) ⟨l, a, r⟩ = ⟨apply_at f ds l, a, r⟩
apply_at f (R # ds) ⟨l, a, r⟩ = ⟨l, a, apply_at f ds r⟩

apply_ats :: ('a tree ⇒ 'a tree) ⇒ pos list ⇒ 'a tree ⇒ 'a tree
apply_ats _ [] t = t
apply_ats f (p # ps) t = apply_ats f ps (apply_at f p t)
```

We are interested in left and right rotations:

$$\mathit{rotR} :: {'a}\ \mathit{tree} \Rightarrow {'a}\ \mathit{tree}$$
$$\mathit{rotR}\ \langle\langle A,\ a,\ B\rangle,\ b,\ C\rangle = \langle A,\ a,\ \langle B,\ b,\ C\rangle\rangle$$

$$\mathit{rotL} :: {'a}\ \mathit{tree} \Rightarrow {'a}\ \mathit{tree}$$
$$\mathit{rotL}\ \langle A,\ a,\ \langle B,\ b,\ C\rangle\rangle = \langle\langle A,\ a,\ B\rangle,\ b,\ C\rangle$$

$$\mathit{rotRs} \equiv \mathit{apply\_ats}\ \mathit{rotR}$$
$$\mathit{rotLs} \equiv \mathit{apply\_ats}\ \mathit{rotL}$$

Now we can prove by computation induction that $\mathit{rotRs}\ (\mathit{rotR\_poss}\ t)$ transforms t into list-form and preserves *inorder*

$$\mathit{is\_list}\ (\mathit{rotRs}\ (\mathit{rotR\_poss}\ t)\ t) \tag{5.9}$$

$$\mathit{inorder}\ (\mathit{rotRs}\ (\mathit{rotR\_poss}\ t)\ t) = \mathit{inorder}\ t \tag{5.10}$$

using the inductive lemma

$$\mathit{apply\_ats}\ f\ (\mathit{map}\ ((\#)\ R)\ ps)\ \langle l,\ a,\ r\rangle = \langle l,\ a,\ \mathit{apply\_ats}\ f\ ps\ r\rangle \tag{5.11}$$

Moreover, we can now express and prove how many right-rotations are required:

$$|\mathit{rotR\_poss}\ t| = |t| - \mathit{rlen}\ t \tag{5.12}$$

The reason: each right-rotation moves one more node onto the right spine. The proof is by computation induction and uses an easy inductive fact: $\mathit{rlen}\ t \le |t|$.

Thus the number of right-rotations to reach list-form is upper-bounded by $|t|$. In fact, (5.12) implies an upper bound of $|t| - 1$ because $|t| - \mathit{rlen}\ t \le |t| - 1$ (why?). This upper bound is tight: any tree with only one node on the right spine needs that many right-rotations because each right-rotation increases *rlen* only by one.

At last we return to the original question, how to transform any tree into any other tree by rotations. The key lemma, which we can express at last, is that reversing the transformation to list-form takes us back to the original tree:

$$\mathit{rotLs}\ (\mathit{rev}\ (\mathit{rotR\_poss}\ t))\ (\mathit{rotRs}\ (\mathit{rotR\_poss}\ t)\ t) = t \tag{5.13}$$

The proof is an easy computation induction using (5.11), the fact that *map* and *rev* commute and the easy inductive fact

$$\mathit{apply\_ats}\ f\ (ps_1\ @\ ps_2)\ t = \mathit{apply\_ats}\ f\ ps_2\ (\mathit{apply\_ats}\ f\ ps_1\ t)$$

With this easy inductive proposition

$$\mathit{is\_list}\ t_1 \wedge \mathit{is\_list}\ t_2 \wedge \mathit{inorder}\ t_1 = \mathit{inorder}\ t_2 \longrightarrow t_1 = t_2 \tag{5.14}$$

we can finally transform any t_1 into any t_2 by rotations if $\mathit{inorder}\ t_1 = \mathit{inorder}\ t_2$. First observe that

$$\mathit{rotRs}\ (\mathit{rotR\_poss}\ t_1)\ t_1 = \mathit{rotRs}\ (\mathit{rotR\_poss}\ t_2)\ t_2$$

follows from $\mathit{inorder}\ t_1 = \mathit{inorder}\ t_2$, (5.9), (5.10) and (5.14). Thus we obtain

$$\begin{aligned}
&\mathit{rotLs}\ (\mathit{rev}\ (\mathit{rotR\_poss}\ t_2))\ (\mathit{rotRs}\ (\mathit{rotR\_poss}\ t_1)\ t_1)\\
&= \mathit{rotLs}\ (\mathit{rev}\ (\mathit{rotR\_poss}\ t_2))\ (\mathit{rotRs}\ (\mathit{rotR\_poss}\ t_2)\ t_2)\\
&= t_2 && \text{by (5.13)}
\end{aligned}$$

5.5.1 Exercises

Exercise 5.5. Define a function *count_rots* that counts the number of right-rotations that *list_of* performs. It should look essentially the same as *list_of* but return the number of rotations rather than the list, similar to a running time function. Prove $\mathit{count\_rots}\ t = |t| - \mathit{rlen}\ t$.

Exercise 5.6. Prove $\exists ps.\ \mathit{is\_list}\ (\mathit{rotRs}\ ps\ t) \wedge \mathit{inorder}\ (\mathit{rotRs}\ ps\ t) = \mathit{inorder}\ t$ by induction, without defining or using a function like *rotR_poss* to compute ps.

Exercise 5.7. Find a tree t and a position list ps such that $\mathit{is\_list}\ (\mathit{rotRs}\ ps\ t)$ and $|ps| > |\mathit{rotR\_poss}\ t|$. Is it possible to rotate a tree into list-form with less than $|t| - \mathit{rlen}\ t$ rotations?

5.6 Case Study: Interval Trees ↗

In this section we study binary trees for representing a set of intervals, called **interval trees**. In addition to the usual insertion and deletion functions of standard BSTs, interval trees support a function for determining whether a given interval overlaps with some interval in the tree.

5.6.1 Augmented BSTs

The efficient implementation of the search for an overlapping interval relies on an additional piece of information in each node. Thus interval trees are another example of augmented trees as introduced in Section 4.4. We reuse the modified definitions of *set_tree* and *inorder* from that section. Moreover we use a slightly adjusted version of *isin* that works for any kind of augmented BST:

```
isin :: ('a × 'b) tree ⇒ 'a ⇒ bool
isin ⟨⟩ _ = False
isin ⟨l, (a, _), r⟩ x
= (case cmp x a of LT ⇒ isin l x | EQ ⇒ True | GT ⇒ isin r x)
```

5.6.2 Intervals

An interval $'a\ ivl$ is simply a pair of lower and upper bound, accessed by functions *low* and *high*, respectively. Intuitively, an interval represents the closed set between *low* and *high*. The standard mathematical notation is $[l,h]$, the Isabelle notation is $\{l..h\}$. We restrict ourselves to non-empty intervals:

$$\mathit{low}\ p \le \mathit{high}\ p$$

Type $'a$ can be any linearly ordered type with a minimum element $\bot$ (for example, the natural numbers or the real numbers extended with $-\infty$). Intervals can be linearly ordered by first comparing *low*, then comparing *high*. The definitions are as follows:

$$(x < y) = (\mathit{low}\ x < \mathit{low}\ y \lor \mathit{low}\ x = \mathit{low}\ y \land \mathit{high}\ x < \mathit{high}\ y)$$
$$(x \le y) = (\mathit{low}\ x < \mathit{low}\ y \lor \mathit{low}\ x = \mathit{low}\ y \land \mathit{high}\ x \le \mathit{high}\ y)$$

Two intervals overlap if they have at least one point in common:

$$\mathit{overlap}\ x\ y = (\mathit{low}\ y \le \mathit{high}\ x \land \mathit{low}\ x \le \mathit{high}\ y)$$

The readers should convince themselves that *overlap* does what it is supposed to do: $\mathit{overlap}\ x\ y = (\{\mathit{low}\ x..\mathit{high}\ x\} \cap \{\mathit{low}\ y..\mathit{high}\ y\} \neq \{\})$

We also define the concept of an interval overlapping with some interval in a set:

$$\mathit{has\_overlap}\ S\ y = (\exists x{\in}S.\ \mathit{overlap}\ x\ y)$$

5.6.3 Interval Trees

An interval tree associates to each node a number *max_hi*, which records the maximum *high* value of all intervals in the subtrees. This value is updated during insert and delete operations, and it will be crucial for enabling efficient determination of overlap with some interval in the tree.

type_synonym $'a\ \mathit{ivl\_tree} = ('a\ \mathit{ivl} \times\ 'a)\ \mathit{tree}$

$$\mathit{max\_hi} :: {'a}\ \mathit{ivl\_tree} \Rightarrow {'a}$$
$$\mathit{max\_hi}\ \langle\rangle = \bot$$
$$\mathit{max\_hi}\ \langle\_,\ (\_,\ m),\ \_\rangle = m$$

If the *max_hi* value of every node in a tree agrees with *max3*

```
inv_max_hi :: 'a ivl_tree ⇒ bool
inv_max_hi ⟨⟩ = True
inv_max_hi ⟨l, (a, m), r⟩
= (m = max3 a l r ∧ inv_max_hi l ∧ inv_max_hi r)

max3 :: 'a ivl ⇒ 'a ivl_tree ⇒ 'a ivl_tree ⇒ 'a
max3 a l r = max (high a) (max (max_hi l) (max_hi r))
```

it follows by induction that *max_hi* is the maximum value of *high* in the tree and comes from some node in the tree:

Lemma 5.1. $\mathit{inv\_max\_hi}\ t \wedge a \in \mathit{set\_tree}\ t \longrightarrow \mathit{high}\ a \leq \mathit{max\_hi}\ t$

Lemma 5.2. $\mathit{inv\_max\_hi}\ t \wedge t \neq \langle\rangle \longrightarrow (\exists\, a {\in} \mathit{set\_tree}\ t.\ \mathit{high}\ a = \mathit{max\_hi}\ t)$

5.6.4 Implementing Sets of Intervals via Interval Trees

Interval trees can implement sets of intervals via unbalanced BSTs as in Section 5.2. Function *isin* was already defined in Section 5.6.1. Insertion and deletion are also very close to the versions in Section 5.2, but the value of *max_hi* must be computed (by *max3*) for each new node. We follow Section 4.4 and introduce a smart constructor *node* for that purpose and replace $\langle l,\ a,\ r\rangle$ by *node* $l\ a\ r$ (on the right-hand side):

```
node :: 'a ivl_tree ⇒ 'a ivl ⇒ 'a ivl_tree ⇒ 'a ivl_tree
node l a r = ⟨l, (a, max3 a l r), r⟩

insert :: 'a ivl ⇒ 'a ivl_tree ⇒ 'a ivl_tree
insert x ⟨⟩ = ⟨⟨⟩, (x, high x), ⟨⟩⟩
insert x ⟨l, (a, m), r⟩ = (case cmp x a of
                           LT ⇒ node (insert x l) a r |
                           EQ ⇒ ⟨l, (a, m), r⟩ |
                           GT ⇒ node l a (insert x r))

split_min :: 'a ivl_tree ⇒ 'a ivl × 'a ivl_tree
split_min ⟨l, (a, _), r⟩
= (if l = ⟨⟩ then (a, r)
   else let (x, l') = split_min l in (x, node l' a r))
```

```
delete :: 'a ivl ⇒ 'a ivl_tree ⇒ 'a ivl_tree
delete _ ⟨⟩ = ⟨⟩
delete x ⟨l, (a, _), r⟩
= (case cmp x a of
    LT ⇒ node (delete x l) a r |
    EQ ⇒ if r = ⟨⟩ then l else let (x, y) = split_min r in node l x y |
    GT ⇒ node l a (delete x r))
```

The correctness proofs for insertion and deletion cover two aspects. Functional correctness and preservation of the invariant $\mathit{sorted} \circ \mathit{inorder}$ (the BST property) are proved exactly as in Section 5.3 for ordinary BSTs. Preservation of the invariant $\mathit{inv\_max\_hi}$ can be proved by a sequence of simple inductive properties. The main correctness properties are these:

$$\mathit{sorted}\ (\mathit{inorder}\ t) \longrightarrow \mathit{inorder}\ (\mathit{insert}\ x\ t) = \mathit{ins\_list}\ x\ (\mathit{inorder}\ t)$$
$$\mathit{sorted}\ (\mathit{inorder}\ t) \longrightarrow \mathit{inorder}\ (\mathit{delete}\ x\ t) = \mathit{del\_list}\ x\ (\mathit{inorder}\ t)$$
$$\mathit{inv\_max\_hi}\ t \longrightarrow \mathit{inv\_max\_hi}\ (\mathit{insert}\ x\ t)$$
$$\mathit{inv\_max\_hi}\ t \longrightarrow \mathit{inv\_max\_hi}\ (\mathit{delete}\ x\ t)$$

Defining $\mathit{invar}\ t = (\mathit{inv\_max\_hi}\ t \wedge \mathit{sorted}\ (\mathit{inorder}\ t))$ we obtain the following top-level correctness corollaries:

$$\mathit{invar}\ s \longrightarrow \mathit{set\_tree}\ (\mathit{insert}\ x\ s) = \mathit{set\_tree}\ s \cup \{x\}$$
$$\mathit{invar}\ s \longrightarrow \mathit{set\_tree}\ (\mathit{delete}\ x\ s) = \mathit{set\_tree}\ s - \{x\}$$
$$\mathit{invar}\ s \longrightarrow \mathit{invar}\ (\mathit{insert}\ x\ s)$$
$$\mathit{invar}\ s \longrightarrow \mathit{invar}\ (\mathit{delete}\ x\ s)$$

The above insertion function allows overlapping intervals to be added into the tree and deletion supports only deletion of whole intervals. This is appropriate for the computational geometry application sketched below in Section 5.6.6. Other applications may require a different design.

5.6.5 Searching for an Overlapping Interval

The added functionality of interval trees over ordinary BSTs is function *search* that searches for an overlapping rather than identical interval:

```
search :: 'a ivl_tree ⇒ 'a ivl ⇒ bool
search ⟨⟩ _ = False
search ⟨l, (a, _), r⟩ x
= (if overlap x a then True
   else if l ≠ ⟨⟩ ∧ low x ≤ max_hi l then search l x else search r x)
```

The following theorem expresses the correctness of *search* assuming the same invariants as before; *bst* t would work just as well as *sorted* (*inorder* t).

Theorem 5.3. *inv_max_hi* $t \wedge$ *sorted* (*inorder* t) $\longrightarrow$
search $t\ x =$ *has_overlap* (*set_tree* t) x

Proof. The result is clear when t is $\langle\rangle$. Now suppose t is in the form $\langle l,\ (a,\ m),\ r\rangle$, where m is the value of *max_hi* at root. If a overlaps with x, search returns *True* as expected. Otherwise, there are two cases.

- If $l \neq \langle\rangle$ and *low* $x \leq$ *max_hi* l, the search goes to the left child. If there is an interval in the left child overlapping with x, then the search returns *True* as expected. Otherwise, we show there is also no interval in the right child overlapping with x. Since $l \neq \langle\rangle$, Lemma 5.2 yields a node p in the left child such that *high* $p =$ *max_hi* l. Since *low* $x \leq$ *max_hi* l, we have *low* $x \leq$ *high* p. Since p does not overlap with x, we must have *high* $x <$ *low* p. But then, for every interval rp in the right child, *low* $p \leq$ *low* rp, so that *high* $x <$ *low* rp, which implies that rp does not overlap with x.
- Now we consider the case where either $l = \langle\rangle$ or *max_hi* $l <$ *low* x. In this case, the search goes to the right. We show there is no interval in the left child that overlaps with x. This is clear if $l = \langle\rangle$. Otherwise, for each interval lp, we have *high* $lp \leq$ *max_hi* l by Lemma 5.1, so that *high* $lp <$ *low* x, which means lp does not overlap with x. □

Exercise 5.8. Define a function that determines if a given point is in some interval in a given interval tree. Starting with

in_ivl :: $'a \Rightarrow\ 'a\ ivl \Rightarrow bool$

in_ivl $x\ iv =$ (*low* $iv \leq x \wedge x \leq$ *high* iv)

write a recursive function

search1 :: $'a\ ivl\_tree \Rightarrow\ 'a \Rightarrow bool$

(without using *search*) such that *search1* $x\ t$ is *True* iff there is some interval iv in t such that *in_ivl* $x\ iv$. Prove

inv_max_hi $t \wedge$ *bst* $t \longrightarrow$ *search1* $t\ x = (\exists iv \in$ *set_tree* $t.$ *in_ivl* $x\ iv)$

5.6.6 Application

While this section demonstrated how to augment an ordinary binary tree with intervals, any of the balanced binary trees (such as red-black tree) can be augmented in a similar manner. We leave this as exercises.

Interval trees have many applications in computational geometry. As a basic example, consider a set of rectangles whose sides are aligned to the x and y-axes. We wish to efficiently determine whether any pair of rectangles in the set intersect each other (i.e. sharing a point, including boundaries). This can be done using a "sweep line" algorithm as follows. For each rectangle $[x_l, x_h] \times [y_l, y_h]$, we create two events: insert interval $[x_l, x_h]$ at y-coordinate y_l and delete interval $[x_l, x_h]$ at y-coordinate y_h. Perform the events, starting from an empty interval tree, in ascending order of y-coordinates, with insertion events performed before deletion events. At each insertion, check whether the interval to be inserted overlaps with any of the existing intervals in the tree. If yes, we have found an intersection between two rectangles. If no overlap of intervals is detected throughout the process, then no pair of rectangles intersect. When using an interval tree based on a balanced binary tree, the time complexity of this procedure is $O(n \lg n)$, where n is the number of rectangles.

Chapter Notes

Tree Rotations and Distance Culík II and Wood [1982] defined the **rotation distance** of two trees t_1 and t_2 with the same number of nodes n as the minimum number of rotations needed to transform t_1 into t_2 and showed that it is upper-bounded by $2n - 2$. This result was improved by Sleator et al. [1986] and Pournin [2014] who showed that for $n \geq 11$ the maximum rotation distance is exactly $2n - 6$. The complexity of computing the rotation distance is open: it is in NP but it is currently not known if it is NP-complete.

Interval Trees We refer to Cormen et al. [2009, Section 14.3] for another exposition on interval trees and their applications. Interval trees, together with the application of finding rectangle intersection, have been formalized by Zhan [2018].

6 Abstract Data Types

Tobias Nipkow

In the previous chapter we looked at a very specific example of an abstract data type, namely sets. In this chapter we consider abstract data types in general and in particular the model-oriented approach to the specification of abstract data types. This will lead to a generic format for such specifications. As a second example we consider the abstract data type of maps.

6.1 Abstract Data Types

Abstract data types (ADTs) can be summarized by the following slogan:

$$\text{ADT} = \textit{interface} + \textit{specification}$$

where the interface lists the operations supported by the ADT and the specification describes the behaviour of these operations. For example, our set ADT has the following interface:

$$\begin{aligned}
&\textit{empty} :: {}'s\\
&\textit{insert} :: {}'a \Rightarrow {}'s \Rightarrow {}'s\\
&\textit{delete} :: {}'a \Rightarrow {}'s \Rightarrow {}'s\\
&\textit{isin} :: {}'s \Rightarrow {}'a \Rightarrow \textit{bool}
\end{aligned}$$

The purpose of an ADT is to be able to write applications based on this ADT that will work with any implementation of the ADT. To this end one can prove properties of the application that are solely based on the specification of the ADT. That is, one can write generic algorithms and prove generic correctness theorems about them in the context of the ADT specification.

6.2 Model-Oriented Specification

We follow the model-oriented style of specification advocated by Jones [1990]. In that style, an abstract type is specified by giving an abstract model for it. For simplicity we assume that each ADT describes one **type of interest** T. In the set interface T is $'s$. This type T must be specified by some existing HOL type A, the abstract model. In the case of sets this is straightforward: the model for sets is simply the HOL type $'a\ set$. The motto is that T should behave like A. In order to bridge the gap between the two types, the specification needs an

- **abstraction function** $\alpha :: T \Rightarrow A$

that maps concrete values to their abstract counterparts. Moreover, in general only some elements of T represent elements of A. For example, in the set implementation in the previous chapter not all trees but only BSTs represent sets. Thus the specification should also take into account an

- **invariant** $\mathit{invar} :: T \Rightarrow \mathit{bool}$

Note that the abstraction function and the invariant are not part of the interface, but they are essential for specification and verification purposes.

As an example, the ADT of sets is shown in Figure 6.1 with suggestive keywords and a fixed mnemonic naming schema for the labels in the specification. This is the template for ADTs that we follow throughout the book. We have intentionally refrained from showing the Isabelle formalization using a so-called **locale** and have opted for a more intuitive textual format that is not Isabelle-specific, in accordance with the general philosophy of this book. The actual Isabelle text can of course be found in the source files, and locales are explained in a dedicated manual [Ballarin].

ADT Set =

interface

$\mathit{empty} :: {'s}$
$\mathit{insert} :: {'a} \Rightarrow {'s} \Rightarrow {'s}$
$\mathit{delete} :: {'a} \Rightarrow {'s} \Rightarrow {'s}$
$\mathit{isin} :: {'s} \Rightarrow {'a} \Rightarrow \mathit{bool}$

abstraction $\mathit{set} :: {'s} \Rightarrow {'a}\ \mathit{set}$
invariant $\mathit{invar} :: {'s} \Rightarrow \mathit{bool}$

specification

$\mathit{set}\ \mathit{empty} = \{\}$ (*empty*)
$\mathit{invar}\ \mathit{empty}$ (*empty-inv*)

$\mathit{invar}\ s \longrightarrow \mathit{set}\ (\mathit{insert}\ x\ s) = \mathit{set}\ s \cup \{x\}$ (*insert*)
$\mathit{invar}\ s \longrightarrow \mathit{invar}\ (\mathit{insert}\ x\ s)$ (*insert-inv*)

$\mathit{invar}\ s \longrightarrow \mathit{set}\ (\mathit{delete}\ x\ s) = \mathit{set}\ s - \{x\}$ (*delete*)
$\mathit{invar}\ s \longrightarrow \mathit{invar}\ (\mathit{delete}\ x\ s)$ (*delete-inv*)

$\mathit{invar}\ s \longrightarrow \mathit{isin}\ s\ x = (x \in \mathit{set}\ s)$ (*isin*)

Figure 6.1 ADT *Set*

We conclude this section by explaining what the specification of an arbitrary ADT looks like. We assume that for each function f of the interface there is a corresponding function f_A in the abstract model A. For a uniform treatment we extend α and *invar* to arbitrary types by setting $\alpha\ x = x$ and $invar\ x =$ *True* for all types other than T. Each function f of the interface gives rise to two properties in the specification: **preservation of the invariant** and simulation of f_A. The precondition is shared:

$$\begin{array}{ll} invar\ x_1 \wedge \ldots \wedge invar\ x_n \longrightarrow & \\ \quad invar(f\ x_1\ \ldots\ x_n) & (f\text{-}inv) \\ \quad \alpha(f\ x_1\ \ldots\ x_n) = f_A\ (\alpha\ x_1)\ \ldots\ (\alpha\ x_n) & (f) \end{array}$$

To understand how the specification of ADT *Set* is the result of this uniform schema one has to take two things into account:

- Precisely which abstract operations on type *'a set* model the functions in the interface of the ADT *Set*? This correspondence is implicit in the specification: *empty* is modeled by $\{\}$, *insert* is modeled by $\lambda x\ s.\ s \cup \{x\}$, *delete* is modeled by $\lambda x\ s.\ s - \{x\}$ and *isin* is modeled by $\lambda s\ x.\ x \in s$.
- Because of the artificial extension of α and *invar* the above uniform format often collapses to something simpler where some α's and *invar*'s disappear.

6.3 Implementing ADTs

An implementation of an ADT consists of definitions for all the functions in the interface. For the correctness proof, you also need to provide an abstraction function and the invariant. The latter two need not be executable unless they also occur in the interface and the implementation is meant to be executable. Finally you need to prove all propositions in the specification of the ADT, of course replacing the function names in the ADT by their implementations.

For Isabelle users: because ADTs are formalized as locales, an implementation of an ADT is an interpretation of the corresponding locale.

Exercise 6.1. Sets of natural numbers can be implemented as lists of intervals, where an interval is simply a pair of numbers. For example, the set {2, 3, 5, 7, 8, 9} can be represented by the list [(2, 3), (5, 5), (7, 9)].

type_synonym *interval* = *nat* × *nat*
type_synonym *intervals* = *interval list*

Define an abstraction function and invariant

set_of :: *intervals* ⇒ *nat set*
invar :: *intervals* ⇒ *bool*

The invariant should enforce that all intervals are non-empty, they are sorted in ascending order and they do not overlap. Then define two functions for adding and deleting numbers to and from *intervals*:

$$
\begin{aligned}
&\mathsf{isin} :: \mathit{intervals} \Rightarrow \mathit{nat} \Rightarrow \mathit{bool}\\
&\mathsf{add1} :: \mathit{nat} \Rightarrow \mathit{intervals} \Rightarrow \mathit{intervals}\\
&\mathsf{del1} :: \mathit{nat} \Rightarrow \mathit{intervals} \Rightarrow \mathit{intervals}
\end{aligned}
$$

Show that $[]$, $\mathsf{add1}$, $\mathsf{del1}$, isin, $\mathsf{set\_of}$ and invar correctly implement the ADT *Set* by proving all propositions in the specification, suitably renamed, e.g. $\mathsf{invar}\ \mathit{ivs} \longrightarrow \mathsf{set\_of}\ (\mathsf{add1}\ i\ \mathit{ivs}) = \mathsf{set\_of}\ \mathit{ivs} \cup \{i\}$.

In a second step, define two functions

$$
\begin{aligned}
&\mathsf{add} :: \mathit{intervals} \Rightarrow \mathit{intervals} \Rightarrow \mathit{intervals}\\
&\mathsf{del} :: \mathit{intervals} \Rightarrow \mathit{intervals} \Rightarrow \mathit{intervals}
\end{aligned}
$$

for union and difference and prove

$$
\begin{aligned}
&\mathsf{invar}\ \mathit{xs} \wedge \mathsf{invar}\ \mathit{ys} \longrightarrow \mathsf{set\_of}\ (\mathsf{add}\ \mathit{xs}\ \mathit{ys}) = \mathsf{set\_of}\ \mathit{xs} \cup \mathsf{set\_of}\ \mathit{ys}\\
&\mathsf{invar}\ \mathit{xs} \wedge \mathsf{invar}\ \mathit{ys} \longrightarrow \mathsf{set\_of}\ (\mathsf{del}\ \mathit{xs}\ \mathit{ys}) = \mathsf{set\_of}\ \mathit{ys} - \mathsf{set\_of}\ \mathit{xs}
\end{aligned}
$$

and that they preserve the invariant.

Make sure all functions in your implementation terminate as soon as possible. Both add and del should take time linear in the sum of the lengths of their arguments. They should not simply iterate $\mathsf{add1}$ and $\mathsf{del1}$.

6.4 Maps

An even more versatile type than sets are maps from $'a$ to $'b$. In fact, sets can be viewed as maps from $'a$ to *bool*. Conversely, many data structures for sets also support maps, e.g. BSTs. Although, for simplicity, we mostly focus on sets in this book, maps are used in a few places too.

Just as with sets, there is both an HOL type of maps and an ADT of maps. We start with the former, where $\rightharpoonup$ is just nice syntax:

$$\textbf{type\_synonym}\ 'a \rightharpoonup\ 'b =\ 'a \Rightarrow\ 'b\ \mathit{option}$$

These maps can also be viewed as partial functions. We define the following abbreviation:

$$m(a \mapsto b) \equiv m(a := \mathsf{Some}\ b)$$

The ADT *Map* is shown in Figure 6.2. Type $'m$ represents the type of maps from $'a$ to $'b$. The ADT *Map* is very similar to the ADT *Set* except that the abstraction function *lookup* is also part of the interface: it abstracts a map to a function of type $'a \rightharpoonup 'b$. This implies that the equations are between functions of that type. We use the function update notation (Section 1.3) to explain *update* and *delete*: *update* is modeled by $\lambda m\ a\ b.\ m(a \mapsto b)$ and *delete* by $\lambda m\ a.\ m(a := \mathsf{None})$.

ADT $Map =$

interface

$empty :: 'm$

$update :: 'a \Rightarrow 'b \Rightarrow 'm \Rightarrow 'm$

$delete :: 'a \Rightarrow 'm \Rightarrow 'm$

$lookup :: 'm \Rightarrow 'a \rightharpoonup 'b$

abstraction *lookup*

invariant $invar :: 'm \Rightarrow bool$

specification

$lookup\ empty = (\lambda\_.\ \mathsf{None})$ (*empty*)

$invar\ empty$ (*empty-inv*)

$invar\ m \longrightarrow lookup\ (update\ a\ b\ m) = (lookup\ m)(a \mapsto b)$ (*update*)

$invar\ m \longrightarrow invar\ (update\ a\ b\ m)$ (*update-inv*)

$invar\ m \longrightarrow lookup\ (delete\ a\ m) = (lookup\ m)(a := \mathsf{None})$ (*delete*)

$invar\ m \longrightarrow invar\ (delete\ a\ m)$ (*delete-inv*)

Figure 6.2 ADT *Map*

6.5 Implementing Maps by BSTs

We implement maps as BSTs of type $('a \times 'b)\ tree$. The interface functions have the following straightforward implementations, ignoring the trivial *empty*:

```
lookup :: ('a × 'b) tree ⇒ 'a ⇀ 'b
lookup ⟨⟩ _ = None
lookup ⟨l, (a, b), r⟩ x = (case cmp x a of
                              LT ⇒ lookup l x |
                              EQ ⇒ Some b |
                              GT ⇒ lookup r x)
```

```
update :: 'a ⇒ 'b ⇒ ('a × 'b) tree ⇒ ('a × 'b) tree
update x y ⟨⟩ = ⟨⟨⟩, (x, y), ⟨⟩⟩
update x y ⟨l, (a, b), r⟩ = (case cmp x a of
                              LT ⇒ ⟨update x y l, (a, b), r⟩ |
                              EQ ⇒ ⟨l, (x, y), r⟩ |
                              GT ⇒ ⟨l, (a, b), update x y r⟩)

delete :: 'a ⇒ ('a × 'b) tree ⇒ ('a × 'b) tree
delete _ ⟨⟩ = ⟨⟩
delete x ⟨l, (a, b), r⟩
= (case cmp x a of
    LT ⇒ ⟨delete x l, (a, b), r⟩ |
    EQ ⇒ if r = ⟨⟩ then l
          else let (ab', r') = split_min r in ⟨l, ab', r'⟩ |
    GT ⇒ ⟨l, (a, b), delete x r⟩)
```

Function *split_min* is the one defined in Section 5.6.4.

The correctness proof proceeds as in Section 5.4. The intermediate level is the type $('a \times 'b)$ *list* of association lists sorted w.r.t. the *fst* component:

```
sorted1 ps ≡ sorted (map fst ps)
```

Functions *update*, *delete* and *lookup* are easily implemented:

```
upd_list :: 'a ⇒ 'b ⇒ ('a × 'b) list ⇒ ('a × 'b) list
upd_list x y [] = [(x, y)]
upd_list x y ((a, b) # ps)
= (if x < a then (x, y) # (a, b) # ps
   else if x = a then (x, y) # ps else (a, b) # upd_list x y ps)

del_list :: 'a ⇒ ('a × 'b) list ⇒ ('a × 'b) list
del_list _ [] = []
del_list x ((a, b) # ps) = (if x = a then ps else (a, b) # del_list x ps)
```

$$\begin{aligned}
&\mathit{map\_of} :: ('a \times 'b)\ \mathit{list} \Rightarrow 'a \rightharpoonup 'b\\
&\mathit{map\_of}\ [] = (\lambda x.\ \mathit{None})\\
&\mathit{map\_of}\ ((a,\ b)\ \#\ ps) = (\mathit{map\_of}\ ps)(a \mapsto b)
\end{aligned}$$

It is easy to prove that association lists implement maps of type $'a \rightharpoonup 'b$ via the abstraction function *map_of*:

$$\mathit{map\_of}\ (\mathit{upd\_list}\ x\ y\ ps) = (\mathit{map\_of}\ ps)(x \mapsto y)$$
$$\mathit{sorted1}\ ps \longrightarrow \mathit{map\_of}\ (\mathit{del\_list}\ x\ ps) = (\mathit{map\_of}\ ps)(x := \mathit{None})$$
$$\mathit{sorted1}\ ps \longrightarrow \mathit{sorted1}\ (\mathit{upd\_list}\ x\ y\ ps)$$
$$\mathit{sorted1}\ ps \longrightarrow \mathit{sorted1}\ (\mathit{del\_list}\ x\ ps)$$

The correctness of *map_of* (as an operation on association lists) is trivial because *map_of* is also the abstraction function and thus the requirement becomes $\mathit{map\_of}\ ps\ a = \mathit{map\_of}\ ps\ a$.

We can also prove that $('a \times 'b)\ \mathit{tree}$s implement association lists:

$$\mathit{sorted1}\ (\mathit{inorder}\ t) \longrightarrow \mathit{inorder}\ (\mathit{update}\ a\ b\ t) = \mathit{upd\_list}\ a\ b\ (\mathit{inorder}\ t)$$
$$\mathit{sorted1}\ (\mathit{inorder}\ t) \longrightarrow \mathit{inorder}\ (\mathit{delete}\ x\ t) = \mathit{del\_list}\ x\ (\mathit{inorder}\ t)$$
$$\mathit{sorted1}\ (\mathit{inorder}\ t) \longrightarrow \mathit{lookup}\ t\ x = \mathit{map\_of}\ (\mathit{inorder}\ t)\ x$$

The *Map* specification properties follow by composing the above two sets of implementation properties.

Exercise 6.2. Modify the ADT *Map* as follows. Replace *update* and *delete* by a single function $\mathit{modify} :: 'a \Rightarrow ('b\ \mathit{option} \Rightarrow 'b\ \mathit{option}) \Rightarrow 'm \Rightarrow 'm$ with the specification that *invar m* implies

$$\mathit{lookup}\ (\mathit{modify}\ a\ f\ m) = (\mathit{lookup}\ m)(a := f\ (\mathit{lookup}\ m\ a))$$
$$\mathit{invar}\ (\mathit{modify}\ a\ f\ m)$$

Define *update* and *delete* with the help of *modify* and prove the *update* and *delete* properties in the original ADT *Map* from these definitions and the specification of *modify*. Conversely, in the context of the original ADT *Map*, define *modify* in terms of *update* and *delete* and prove the above properties.

7 2-3 Trees

Tobias Nipkow

This is the first in a series of chapters examining **balanced search trees** where the height of the tree is logarithmic in its size and which can therefore be searched in logarithmic time.

The most popular first example of balanced search trees are red-black trees. We start with **2-3 trees**, where nodes can have 2 or 3 children, because red-black trees are best understood as an implementation of (a variant of) 2-3 trees. We introduce red-black trees in the next chapter. The type of 2-3 trees is similar to binary trees but with an additional constructor *Node3*:

```
datatype 'a tree23 =
  Leaf |
  Node2 ('a tree23) 'a ('a tree23) |
  Node3 ('a tree23) 'a ('a tree23) 'a ('a tree23)
```

The familiar syntactic sugar is sprinkled on top:

$$\begin{aligned}
\langle\rangle &\equiv \mathit{Leaf}\\
\langle l,\ a,\ r\rangle &\equiv \mathit{Node2}\ l\ a\ r\\
\langle l,\ a,\ m,\ b,\ r\rangle &\equiv \mathit{Node3}\ l\ a\ m\ b\ r
\end{aligned}$$

The size, height and the completeness of a 2-3 tree are defined by adding an equation for *Node3* to the corresponding definitions on binary trees:

$$|\langle l,\ \_,\ m,\ \_,\ r\rangle| = |l| + |m| + |r| + 1$$

$$h\ \langle l,\ \_,\ m,\ \_,\ r\rangle = \mathit{max}\ (h\ l)\ (\mathit{max}\ (h\ m)\ (h\ r)) + 1$$

$$\begin{aligned}
&\mathit{complete}\ \langle l,\ \_,\ m,\ \_,\ r\rangle\\
&= (h\ l = h\ m \wedge h\ m = h\ r \wedge \mathit{complete}\ l \wedge \mathit{complete}\ m \wedge \mathit{complete}\ r)
\end{aligned}$$

A trivial induction yields $\mathit{complete}\ t \longrightarrow 2^{h\ t} \le |t| + 1$: thus all operations on complete 2-3 trees have logarithmic complexity if they descend along a single branch and take constant time per node. This is the case and we will not discuss complexity in any more detail.

A nice property of 2-3 trees is that for every n there is a complete 2-3 tree of size n. As we will see below, completeness can be maintained under insertion and deletion in logarithmic time.

Exercise 7.1. Define a function $\mathit{maxt} :: \mathit{nat} \Rightarrow \mathit{unit}\ \mathit{tree}23$ that creates the tree with the largest number of nodes given the height of the tree. We use type *unit* because we are not interested in the elements in the tree. Prove $|\mathit{maxt}\ n| = (3^n - 1)\ \mathrm{div}\ 2$ and that no tree of the given height can be larger: $|t| \le (3^{h\ t} - 1)\ \mathrm{div}\ 2$. Note that both subtraction and division on type *nat* can be tedious to work with. You may want to prove the two properties as corollaries of subtraction- and division-free properties. Alternatively, work with *real* instead of *nat* by replacing *div* by /.

7.1 Implementation of ADT *Set*

The implementation will maintain the usual ordering invariant and completeness. When we speak of a 2-3 tree we will implicitly assume these two invariants now.

Searching a 2-3 tree is like searching a binary tree (see Section 5.2) but with one more defining equation:

```
isin ⟨l, a, m, b, r⟩ x
= (case cmp x a of LT ⇒ isin l x | EQ ⇒ True
   | GT ⇒ case cmp x b of LT ⇒ isin m x | EQ ⇒ True | GT ⇒ isin r x)
```

Insertion into a 2-3 tree must preserve completeness. Thus recursive calls must report back if the tree has increased in height (Of = "overflow") or if the height has stayed the same (Eq_i). Therefore insertion returns a result of this type:

```
datatype 'a up_i = Eq_i ('a tree23) | Of ('a tree23) 'a ('a tree23)
```

This is the idea: If insertion into t returns

$\mathit{Eq}_i\ t'$ then t' has the same height as t,
$\mathit{Of}\ l\ x\ r$ then l and r have the same height as t.

The insertion functions are shown in Figure 7.1. The actual work is performed by the recursive function *ins*. The element to be inserted is propagated down to a leaf, which causes an overflow of the leaf. If an overflow is returned from a recursive call

```
insert x t = tree_i (ins x t)

ins :: 'a ⇒ 'a tree23 ⇒ 'a up_i

ins x ⟨⟩ = Of ⟨⟩ x ⟨⟩
ins x ⟨l, a, r⟩ = (case cmp x a of
                   LT ⇒ case ins x l of
                          Eq_i l' ⇒ Eq_i ⟨l', a, r⟩ |
                          Of l_1 b l_2 ⇒ Eq_i ⟨l_1, b, l_2, a, r⟩ |
                   EQ ⇒ Eq_i ⟨l, a, r⟩ |
                   GT ⇒ case ins x r of
                          Eq_i r' ⇒ Eq_i ⟨l, a, r'⟩ |
                          Of r_1 b r_2 ⇒ Eq_i ⟨l, a, r_1, b, r_2⟩)
ins x ⟨l, a, m, b, r⟩
= (case cmp x a of
    LT ⇒ case ins x l of
           Eq_i l' ⇒ Eq_i ⟨l', a, m, b, r⟩ |
           Of l_1 c l_2 ⇒ Of ⟨l_1, c, l_2⟩ a ⟨m, b, r⟩ |
    EQ ⇒ Eq_i ⟨l, a, m, b, r⟩ |
    GT ⇒ case cmp x b of
           LT ⇒ case ins x m of
                  Eq_i m' ⇒ Eq_i ⟨l, a, m', b, r⟩ |
                  Of m_1 c m_2 ⇒ Of ⟨l, a, m_1⟩ c ⟨m_2, b, r⟩ |
           EQ ⇒ Eq_i ⟨l, a, m, b, r⟩ |
           GT ⇒ case ins x r of
                  Eq_i r' ⇒ Eq_i ⟨l, a, m, b, r'⟩ |
                  Of r_1 c r_2 ⇒ Of ⟨l, a, m⟩ b ⟨r_1, c, r_2⟩)
```

Figure 7.1 Insertion into 2-3 tree

it can be absorbed into a *Node2* but in a *Node3* it causes another overflow. At the root of the tree, function *tree*$_i$ converts values of type up_i back into trees:

```
tree_i :: 'a up_i ⇒ 'a tree23
tree_i (Eq_i t) = t
tree_i (Of l a r) = ⟨l, a, r⟩
```

Deletion is dual. Recursive calls must report back to the caller if the child has “underflown”, i.e. decreased in height. Therefore deletion returns a result of this type:

datatype $'a\ up_d = \mathit{Eq}_d\ ('a\ tree23)\ |\ \mathit{Uf}\ ('a\ tree23)$

This is the idea: If deletion from t returns

$\mathit{Eq}_d\ t'$	then t' has the same height as t,
$\mathit{Uf}\ t'$	then t' is one level lower than t.

The main deletion functions are shown in Figure 7.2. The actual work is performed by the recursive function *del*. If the element to be deleted is in a child, the result of a recursive call is reintegrated into the node via the auxiliary functions $\mathit{node}ij$ from Figure 7.3: $\mathit{node}ij$ creates a node with i children, where child j is given as an up_d value, and wraps the node up in *Uf* or Eq_d, depending on whether an underflow occurred or not. If the element to be deleted is in the node itself, a replacement is obtained and deleted from a child via *split_min*. At the root of the tree, up_d values are converted back into trees:

$\mathit{tree}_d :: 'a\ up_d \Rightarrow 'a\ tree23$

$\mathit{tree}_d\ (\mathit{Eq}_d\ t) = t$
$\mathit{tree}_d\ (\mathit{Uf}\ t) = t$

7.2 Preservation of Completeness

As explained in Section 5.4, we do not go into the automatic functional correctness proofs but concentrate on invariant preservation. To express the relationship between the height of a tree before and after insertion we define a height function h_i:

$h_i :: 'a\ up_i \Rightarrow nat$

$h_i\ (\mathit{Eq}_i\ t) = h\ t$
$h_i\ (\mathit{Of}\ l\ \_\ \_) = h\ l$

Intuitively, h_i is the height of the tree *before* insertion. A routine induction proves

$$\mathit{complete}\ t \longrightarrow \mathit{complete}\ (\mathit{tree}_i\ (\mathit{ins}\ a\ t)) \wedge h_i\ (\mathit{ins}\ a\ t) = h\ t$$

which implies by definition that

$$\mathit{complete}\ t \longrightarrow \mathit{complete}\ (\mathit{insert}\ a\ t)$$

```
delete :: 'a ⇒ 'a tree23 ⇒ 'a tree23
delete x t = tree_d (del x t)

del :: 'a ⇒ 'a tree23 ⇒ 'a up_d
del _ ⟨⟩ = Eq_d ⟨⟩
del x ⟨⟨⟩, a, ⟨⟩⟩ = (if x = a then Uf ⟨⟩ else Eq_d ⟨⟨⟩, a, ⟨⟩⟩)
del x ⟨⟨⟩, a, ⟨⟩, b, ⟨⟩⟩
= Eq_d (if x = a then ⟨⟨⟩, b, ⟨⟩⟩
          else if x = b then ⟨⟨⟩, a, ⟨⟩⟩ else ⟨⟨⟩, a, ⟨⟩, b, ⟨⟩⟩)
del x ⟨l, a, r⟩
= (case cmp x a of LT ⇒ node21 (del x l) a r
    | EQ ⇒ let (a', r') = split_min r in node22 l a' r'
    | GT ⇒ node22 l a (del x r))
del x ⟨l, a, m, b, r⟩
= (case cmp x a of LT ⇒ node31 (del x l) a m b r
    | EQ ⇒ let (a', m') = split_min m in node32 l a' m' b r
    | GT ⇒ case cmp x b of LT ⇒ node32 l a (del x m) b r
               | EQ ⇒ let (b', r') = split_min r in node33 l a m b' r'
               | GT ⇒ node33 l a m b (del x r))

split_min :: 'a tree23 ⇒ 'a × 'a up_d
split_min ⟨⟨⟩, a, ⟨⟩⟩ = (a, Uf ⟨⟩)
split_min ⟨⟨⟩, a, ⟨⟩, b, ⟨⟩⟩ = (a, Eq_d ⟨⟨⟩, b, ⟨⟩⟩)
split_min ⟨l, a, r⟩ = (let (x, l') = split_min l in (x, node21 l' a r))
split_min ⟨l, a, m, b, r⟩
= (let (x, l') = split_min l in (x, node31 l' a m b r))
```

Figure 7.2 Deletion from 2-3 tree: main functions

$$\begin{array}{l}
\mathit{node21} :: {'}a\ \mathit{up}_d \Rightarrow {'}a \Rightarrow {'}a\ \mathit{tree23} \Rightarrow {'}a\ \mathit{up}_d \\
\mathit{node21}\ (\mathit{Eq}_d\ t_1)\ a\ t_2 = \mathit{Eq}_d\ \langle t_1,\ a,\ t_2\rangle \\
\mathit{node21}\ (\mathit{Uf}\ t_1)\ a\ \langle t_2,\ b,\ t_3\rangle = \mathit{Uf}\ \langle t_1,\ a,\ t_2,\ b,\ t_3\rangle \\
\mathit{node21}\ (\mathit{Uf}\ t_1)\ a\ \langle t_2,\ b,\ t_3,\ c,\ t_4\rangle = \mathit{Eq}_d\ \langle\langle t_1,\ a,\ t_2\rangle,\ b,\ \langle t_3,\ c,\ t_4\rangle\rangle
\end{array}$$

$$\begin{array}{l}
\mathit{node22} :: {'}a\ \mathit{tree23} \Rightarrow {'}a \Rightarrow {'}a\ \mathit{up}_d \Rightarrow {'}a\ \mathit{up}_d \\
\mathit{node22}\ t_1\ a\ (\mathit{Eq}_d\ t_2) = \mathit{Eq}_d\ \langle t_1,\ a,\ t_2\rangle \\
\mathit{node22}\ \langle t_1,\ b,\ t_2\rangle\ a\ (\mathit{Uf}\ t_3) = \mathit{Uf}\ \langle t_1,\ b,\ t_2,\ a,\ t_3\rangle \\
\mathit{node22}\ \langle t_1,\ b,\ t_2,\ c,\ t_3\rangle\ a\ (\mathit{Uf}\ t_4) = \mathit{Eq}_d\ \langle\langle t_1,\ b,\ t_2\rangle,\ c,\ \langle t_3,\ a,\ t_4\rangle\rangle
\end{array}$$

$$\begin{array}{l}
\mathit{node31} :: {'}a\ \mathit{up}_d \Rightarrow {'}a \Rightarrow {'}a\ \mathit{tree23} \Rightarrow {'}a \Rightarrow {'}a\ \mathit{tree23} \Rightarrow {'}a\ \mathit{up}_d \\
\mathit{node31}\ (\mathit{Eq}_d\ t_1)\ a\ t_2\ b\ t_3 = \mathit{Eq}_d\ \langle t_1,\ a,\ t_2,\ b,\ t_3\rangle \\
\mathit{node31}\ (\mathit{Uf}\ t_1)\ a\ \langle t_2,\ b,\ t_3\rangle\ c\ t_4 = \mathit{Eq}_d\ \langle\langle t_1,\ a,\ t_2,\ b,\ t_3\rangle,\ c,\ t_4\rangle \\
\mathit{node31}\ (\mathit{Uf}\ t_1)\ a\ \langle t_2,\ b,\ t_3,\ c,\ t_4\rangle\ d\ t_5 \\
= \mathit{Eq}_d\ \langle\langle t_1,\ a,\ t_2\rangle,\ b,\ \langle t_3,\ c,\ t_4\rangle,\ d,\ t_5\rangle
\end{array}$$

$$\begin{array}{l}
\mathit{node32} :: {'}a\ \mathit{tree23} \Rightarrow {'}a \Rightarrow {'}a\ \mathit{up}_d \Rightarrow {'}a \Rightarrow {'}a\ \mathit{tree23} \Rightarrow {'}a\ \mathit{up}_d \\
\mathit{node32}\ t_1\ a\ (\mathit{Eq}_d\ t_2)\ b\ t_3 = \mathit{Eq}_d\ \langle t_1,\ a,\ t_2,\ b,\ t_3\rangle \\
\mathit{node32}\ t_1\ a\ (\mathit{Uf}\ t_2)\ b\ \langle t_3,\ c,\ t_4\rangle = \mathit{Eq}_d\ \langle t_1,\ a,\ \langle t_2,\ b,\ t_3,\ c,\ t_4\rangle\rangle \\
\mathit{node32}\ t_1\ a\ (\mathit{Uf}\ t_2)\ b\ \langle t_3,\ c,\ t_4,\ d,\ t_5\rangle \\
= \mathit{Eq}_d\ \langle t_1,\ a,\ \langle t_2,\ b,\ t_3\rangle,\ c,\ \langle t_4,\ d,\ t_5\rangle\rangle
\end{array}$$

$$\begin{array}{l}
\mathit{node33} :: {'}a\ \mathit{tree23} \Rightarrow {'}a \Rightarrow {'}a\ \mathit{tree23} \Rightarrow {'}a \Rightarrow {'}a\ \mathit{up}_d \Rightarrow {'}a\ \mathit{up}_d \\
\mathit{node33}\ t_1\ a\ t_2\ b\ (\mathit{Eq}_d\ t_3) = \mathit{Eq}_d\ \langle t_1,\ a,\ t_2,\ b,\ t_3\rangle \\
\mathit{node33}\ t_1\ a\ \langle t_2,\ b,\ t_3\rangle\ c\ (\mathit{Uf}\ t_4) = \mathit{Eq}_d\ \langle t_1,\ a,\ \langle t_2,\ b,\ t_3,\ c,\ t_4\rangle\rangle \\
\mathit{node33}\ t_1\ a\ \langle t_2,\ b,\ t_3,\ c,\ t_4\rangle\ d\ (\mathit{Uf}\ t_5) \\
= \mathit{Eq}_d\ \langle t_1,\ a,\ \langle t_2,\ b,\ t_3\rangle,\ c,\ \langle t_4,\ d,\ t_5\rangle\rangle
\end{array}$$

Figure 7.3 Deletion from 2-3 tree: auxiliary functions

To express the relationship between the height of a tree before and after deletion we define

$$h_d :: {}'a\ up_d \Rightarrow nat$$
$$h_d\ (Eq_d\ t) = h\ t$$
$$h_d\ (Uf\ t) = h\ t + 1$$

The intuition is that h_d is the height of the tree *before* deletion.

We now list a sequence of simple inductive properties that build on each other and culminate in completeness preservation of *delete*:

$$complete\ r \wedge complete\ (tree_d\ l') \wedge h\ r = h_d\ l' \longrightarrow$$
$$complete\ (tree_d\ (node21\ l'\ a\ r))$$
$$0 < h\ r \longrightarrow h_d\ (node21\ l'\ a\ r) = max\ (h_d\ l')\ (h\ r) + 1$$
$$split\_min\ t = (x,\ t') \wedge 0 < h\ t \wedge complete\ t \longrightarrow h_d\ t' = h\ t$$
$$split\_min\ t = (x,\ t') \wedge complete\ t \wedge 0 < h\ t \longrightarrow complete\ (tree_d\ t')$$
$$complete\ t \longrightarrow h_d\ (del\ x\ t) = h\ t$$
$$complete\ t \longrightarrow complete\ (tree_d\ (del\ x\ t))$$
$$complete\ t \longrightarrow complete\ (delete\ x\ t)$$

For each property of *node21* there are analogues properties for the other *nodeij* functions which we omit.

7.3 Converting a List into a 2-3 Tree

We consider the problem of converting a list of elements into a 2-3 tree. If the resulting tree should be a search tree, there is the obvious approach: insert the elements one by one starting from the empty tree. This takes time $\Theta(n \lg n)$. This holds for any data structure where insertion takes time proportional to $\lg n$. In that case inserting n elements one by one takes time proportional to $\lg 1 + \cdots + \lg n = \lg(n!)$. Now $n! \leq n^n$ implies $\lg(n!) \leq n \lg n$. On the other hand, $n^n \leq (n \cdot 1) \cdot ((n-1) \cdot 2) \cdots (1 \cdot n) = (n!)^2$ implies $\frac{1}{2} n \lg n \leq \lg(n!)$. Thus $\lg(n!) \in \Theta(n \lg n)$ (which also follows from Stirling's formula). We have intentionally proved a Θ property because the O property is obvious but one might hope that $\lg 1 + \cdots + \lg n$ has a lower order of growth than $n \lg n$. However, since a search tree can be converted into a sorted list in linear time, the conversion into the search tree cannot be faster than sorting.

Now we turn to the actual topic of this section: converting a list xs into a 2-3 tree t such that *inorder* $t = xs$ — in linear time. Thus we can take advantage of situations where we already know that xs is sorted. The bottom-up conversion algorithm is particularly intuitive. It repeatedly passes over an alternating list $t_1,e_1,t_2,e_2,\ldots,t_k$ of trees and elements, combining trees and elements into new trees. Given elements $a_1,\ldots,a_n$ we start with the alternating list $\langle\rangle,a_1,\langle\rangle,a_2,\ldots,a_n,\langle\rangle$. On every pass over this list, we replace adjacent triples t,a,t' by $\langle t,\ a,\ t'\rangle$, possibly creating a 3-node instead of a 2-node at the end of the list. Once a single tree is left over, we terminate.

We define this type of alternating (and non-empty) lists as a new data type:

```
datatype 'a tree23s = T ('a tree23) | TTs ('a tree23) 'a ('a tree23s)
```

The following examples demonstrate the encoding of alternating lists as terms of type *'a tree23s*:

Alternating list:	t_1	t_1,e_1,t_2	t_1,e_1,t_2,e_2,ts
Encoding:	$T\ t_1$	$TTs\ t_1\ e_1\ (T\ t_2)$	$TTs\ t_1\ e_1\ (TTs\ t_2\ e_2\ ts)$

We also need the following auxiliary functions:

```
len :: 'a tree23s ⇒ nat
len (T _) = 1
len (TTs _ _ ts) = len ts + 1

trees :: 'a tree23s ⇒ 'a tree23 set
trees (T t) = {t}
trees (TTs t _ ts) = {t} ∪ trees ts

inorder2 :: 'a tree23s ⇒ 'a list
inorder2 (T t) = inorder t
inorder2 (TTs t a ts) = inorder t @ a # inorder2 ts
```

Repeatedly passing over the alternating list until only a single tree remains is expressed by the following functions:

```
join_all :: 'a tree23s ⇒ 'a tree23
join_all (T t) = t
join_all ts = join_all (join_adj ts)
```

$$\mathit{join\_adj} :: {}'a\ \mathit{tree23s} \Rightarrow {}'a\ \mathit{tree23s}$$

$$\begin{aligned}
&\mathit{join\_adj}\ (\mathit{TTs}\ t_1\ a\ (\mathit{T}\ t_2)) = \mathit{T}\ \langle t_1,\ a,\ t_2\rangle\\
&\mathit{join\_adj}\ (\mathit{TTs}\ t_1\ a\ (\mathit{TTs}\ t_2\ b\ (\mathit{T}\ t_3))) = \mathit{T}\ \langle t_1,\ a,\ t_2,\ b,\ t_3\rangle\\
&\mathit{join\_adj}\ (\mathit{TTs}\ t_1\ a\ (\mathit{TTs}\ t_2\ b\ ts)) = \mathit{TTs}\ \langle t_1,\ a,\ t_2\rangle\ b\ (\mathit{join\_adj}\ ts)
\end{aligned}$$

Note that *join_adj* is not and does not need to be defined on single trees. We express this precondition with an abbreviation:

$$\mathit{not\_T}\ ts \equiv \nexists t.\ ts = \mathit{T}\ t$$

Also note that *join_all* terminates only because *join_adj* shortens the list:

$$\mathit{not\_T}\ ts \longrightarrow \mathit{len}\ (\mathit{join\_adj}\ ts) < \mathit{len}\ ts$$

In fact, it reduces the length at least by a factor of 2:

$$\mathit{not\_T}\ ts \longrightarrow \mathit{len}\ (\mathit{join\_adj}\ ts) \le \mathit{len}\ ts\ \mathrm{div}\ 2 \tag{7.1}$$

The whole process starts with a list of alternating leaves and elements:

$$\mathit{tree23\_of\_list} :: {}'a\ \mathit{list} \Rightarrow {}'a\ \mathit{tree23}$$

$$\mathit{tree23\_of\_list}\ as = \mathit{join\_all}\ (\mathit{leaves}\ as)$$

$$\mathit{leaves} :: {}'a\ \mathit{list} \Rightarrow {}'a\ \mathit{tree23s}$$

$$\begin{aligned}
&\mathit{leaves}\ [] = \mathit{T}\ \langle\rangle\\
&\mathit{leaves}\ (a\ \#\ as) = \mathit{TTs}\ \langle\rangle\ a\ (\mathit{leaves}\ as)
\end{aligned}$$

7.3.1 Correctness

Functional correctness is easily established. The *inorder* and the completeness properties are proved independently by the following inductive lemmas:

$$\mathit{not\_T}\ ts \longrightarrow \mathit{inorder2}\ (\mathit{join\_adj}\ ts) = \mathit{inorder2}\ ts$$

$$\mathit{inorder}\ (\mathit{join\_all}\ ts) = \mathit{inorder2}\ ts$$

$$\mathit{inorder}\ (\mathit{tree23\_of\_list}\ as) = as$$

$$\begin{aligned}
&(\forall t{\in}\mathit{trees}\ ts.\ \mathit{complete}\ t \wedge \mathit{h}\ t = n) \wedge \mathit{not\_T}\ ts \longrightarrow\\
&(\forall t{\in}\mathit{trees}\ (\mathit{join\_adj}\ ts).\ \mathit{complete}\ t \wedge \mathit{h}\ t = n + 1)
\end{aligned}$$

$$(\forall t{\in}\mathit{trees}\ ts.\ \mathit{complete}\ t \wedge \mathit{h}\ t = n) \longrightarrow \mathit{complete}\ (\mathit{join\_all}\ ts)$$

$$t \in \mathit{trees}\ (\mathit{leaves}\ as) \longrightarrow \mathit{complete}\ t \wedge h\ t = 0$$

$$\mathit{complete}\ (\mathit{tree23\_of\_list}\ as)$$

7.3.2 Running Time

Why does the conversion take linear time? Because the first pass over an alternating list of length n takes n steps, the next pass $n/2$ steps, the next pass $n/4$ steps, etc., and this sums up to $2n$. The time functions for the formal proof are shown in Appendix B.3. The following upper bound is easily proved by induction on the computation of *join_adj*:

$$\mathit{not\_T}\ ts \longrightarrow T_{\mathit{join\_adj}}\ ts \le \mathit{len}\ ts \text{ div } 2 \tag{7.2}$$

An upper bound $T_{\mathit{join\_all}}\ ts \le 2 \cdot \mathit{len}\ ts$ follows by induction on the computation of *join_adj*. We focus on the induction step:

$$\begin{aligned}
& T_{\mathit{join\_all}}\ ts \\
&= T_{\mathit{join\_adj}}\ ts + T_{\mathit{join\_all}}\ (\mathit{join\_adj}\ ts) + 1 \\
&\le \mathit{len}\ ts \text{ div } 2 + 2 \cdot \mathit{len}\ (\mathit{join\_adj}\ ts) + 1 && \text{using (7.2) and IH} \\
&\le \mathit{len}\ ts \text{ div } 2 + 2 \cdot (\mathit{len}\ ts \text{ div } 2) + 1 && \text{by (7.1)} \\
&\le 2 \cdot \mathit{len}\ ts && \text{because } 1 \le \mathit{len}\ ts
\end{aligned}$$

Now it is routine to derive

$$T_{\mathit{tree23\_of\_list}}\ as \le 3 \cdot |as| + 3$$

Chapter Notes

The invention of 2-3 trees is credited to Hopcroft in 1970 by Cormen et al. [2009, p. 337]. Equational definitions were given by Hoffmann and O'Donnell [1982] (only insertion) and Reade [1992]. Our formalisation is based on teaching material by Franklyn Turbak and the article by Hinze [2018].

8 Red-Black Trees

Tobias Nipkow

Red-black trees are a popular implementation technique for BSTs: they guarantee logarithmic height just like 2-3 trees but the code is arguably simpler. The nodes are colored either red or black. Abstractly, red-black trees encode 2-3-4 trees where nodes have between 2 and 4 children. Each 2-3-4 node is encoded by a group of 2, 3 or 4 colored binary nodes as follows:

$$\begin{array}{rcl} \langle\rangle & \approx & \langle\rangle \\ \langle A,a,B\rangle & \approx & \langle A,a,B\rangle \\ \langle A,a,B,b,C\rangle & \approx & \langle\langle A,a,B\rangle,b,C\rangle \text{ or } \langle A,a,\langle B,b,C\rangle\rangle \\ \langle A,a,B,b,C,c,D\rangle & \approx & \langle\langle A,a,B\rangle,b,\langle C,c,D\rangle\rangle \end{array}$$

Color expresses grouping: a black node is the root of a 2-3-4 node, a red node is part of a bigger 2-3-4 node. Thus a red-black tree needs to satisfy the following properties or invariants:

1. The root is black.
2. Every $\langle\rangle$ is considered black.
3. If a node is red, its children are black.
4. All paths from a node to a leaf have the same number of black nodes.

The final property expresses that the corresponding 2-3-4 tree is complete. The last two properties imply that the tree has logarithmic height (see below).

We implement red-black trees as binary trees augmented (see Section 4.4) with a color tag:

```
datatype color = Red | Black

type_synonym 'a rbt = ('a × color) tree
```

Some new syntactic sugar is sprinkled on top:

```
R l a r  ≡  ⟨l, (a, Red), r⟩
B l a r  ≡  ⟨l, (a, Black), r⟩
```

The following functions get and set the color of a node:

```
color :: 'a rbt ⇒ color
color ⟨⟩ = Black
color ⟨_, (_, c), _⟩ = c

paint :: color ⇒ 'a rbt ⇒ 'a rbt
paint _ ⟨⟩ = ⟨⟩
paint c ⟨l, (a, _), r⟩ = ⟨l, (a, c), r⟩
```

Note that the *color* of a leaf is by definition black.

8.1 Invariants

The above informal description of the red-black tree invariants is formalized as the predicate *rbt* which (for reasons of modularity) is split into a color and a height invariant *invc* and *invh*:

```
rbt :: 'a rbt ⇒ bool
rbt t = (invc t ∧ invh t ∧ color t = Black)
```

The color invariant expresses that red nodes must have black children:

```
invc :: 'a rbt ⇒ bool
invc ⟨⟩ = True
invc ⟨l, (_, c), r⟩
= ((c = Red ⟶ color l = Black ∧ color r = Black) ∧
   invc l ∧ invc r)
```

The height invariant expresses (via the **black height** *bh*) that all paths from the root to a leaf have the same number of black nodes:

$$\mathit{invh} :: {}'a\ \mathit{rbt} \Rightarrow \mathit{bool}$$
$$\mathit{invh}\ \langle\rangle = \mathit{True}$$
$$\mathit{invh}\ \langle l,\ (\_,\ \_),\ r\rangle = (\mathit{bh}\ l = \mathit{bh}\ r \wedge \mathit{invh}\ l \wedge \mathit{invh}\ r)$$

$$\mathit{bh} :: {}'a\ \mathit{rbt} \Rightarrow \mathit{nat}$$
$$\mathit{bh}\ \langle\rangle = 0$$
$$\mathit{bh}\ \langle l,\ (\_,\ c),\ \_\rangle = (\textbf{if}\ c = \mathit{Black}\ \textbf{then}\ \mathit{bh}\ l + 1\ \textbf{else}\ \mathit{bh}\ l)$$

Note that although *bh* traverses only the left spine of the tree, the fact that *invh* traverses the complete tree ensures that all paths from the root to a leaf are considered (see Exercise 8.2).

The split of the invariant into *invc* and *invh* improves modularity: frequently one can prove preservation of *invc* and *invh* separately, which facilitates proof search. For compactness we will mostly present the combined invariance properties.

8.1.1 Logarithmic Height

In a red-black tree, i.e. *rbt* t, every path from the root to a leaf has the same number of black nodes, and no such path has two red nodes in a row. In the worst case, there is one path where black and red alternate and all other nodes are black. Then the height is $2 \cdot n$ but the minimal height only n. Using $2^{\mathit{mh}\ t} \leq |t|_1$ this implies $h\ t = 2 \cdot n = 2 \cdot \mathit{lg}\ |t|_1$. Formally: if *rbt* t then

$$h\ t \leq 2 \cdot \mathit{bh}\ t \leq 2 \cdot \mathit{mh}\ t \leq 2 \cdot \mathit{lg}\ |t|_1$$

where the first and second step are corollaries of the following inductive propositions:

$$\mathit{invc}\ t \wedge \mathit{invh}\ t \longrightarrow h\ t \leq 2 \cdot \mathit{bh}\ t + (\textbf{if}\ \mathit{color}\ t = \mathit{Black}\ \textbf{then}\ 0\ \textbf{else}\ 1)$$
$$\mathit{invh}\ t \longrightarrow \mathit{bh}\ t \leq \mathit{mh}\ t$$

8.2 Implementation of ADT *Set*

We implement sets by red-black trees that are also BSTs. As usual, we only discuss the proofs of preservation of the *rbt* invariant.

Function *isin* is implemented as for all augmented BSTs (see Section 5.6.1).

8.2.1 Insertion

Insertion is shown in Figure 8.1. The workhorse is function *ins*. It descends to the leaf where the element is inserted and it adjusts the colors on the way back up. The adjustment is performed by *baliL*/*baliR*. They combine arguments $l\ a\ r$ into a tree. If there is a red-red conflict in l/r, they rebalance and replace it by red-black. Inserting

```
insert x t = paint Black (ins x t)

ins :: 'a ⇒ 'a rbt ⇒ 'a rbt
ins x ⟨⟩ = R ⟨⟩ x ⟨⟩
ins x (B l a r) = (case cmp x a of
                    LT ⇒ baliL (ins x l) a r |
                    EQ ⇒ B l a r |
                    GT ⇒ baliR l a (ins x r))
ins x (R l a r) = (case cmp x a of
                    LT ⇒ R (ins x l) a r |
                    EQ ⇒ R l a r |
                    GT ⇒ R l a (ins x r))

baliL :: 'a rbt ⇒ 'a ⇒ 'a rbt ⇒ 'a rbt
baliL (R (R t₁ a t₂) b t₃) c t₄ = R (B t₁ a t₂) b (B t₃ c t₄)
baliL (R t₁ a (R t₂ b t₃)) c t₄ = R (B t₁ a t₂) b (B t₃ c t₄)
baliL t₁ a t₂ = B t₁ a t₂

baliR :: 'a rbt ⇒ 'a ⇒ 'a rbt ⇒ 'a rbt
baliR t₁ a (R t₂ b (R t₃ c t₄)) = R (B t₁ a t₂) b (B t₃ c t₄)
baliR t₁ a (R (R t₂ b t₃) c t₄) = R (B t₁ a t₂) b (B t₃ c t₄)
baliR t₁ a t₂ = B t₁ a t₂
```

Figure 8.1 Insertion into red-black tree

into a red node needs no immediate balancing because that will happen at the black node above it, for example:

$$
\begin{aligned}
&\mathit{ins}\ 1\ (B\ (R\ \langle\rangle\ 0\ \langle\rangle)\ 2\ (R\ \langle\rangle\ 3\ \langle\rangle))\\
&= \mathit{baliL}\ (\mathit{ins}\ 1\ (R\ \langle\rangle\ 0\ \langle\rangle))\ 2\ (R\ \langle\rangle\ 3\ \langle\rangle)\\
&= \mathit{baliL}\ (R\ \langle\rangle\ 0\ (\mathit{ins}\ 1\ \langle\rangle))\ 2\ (R\ \langle\rangle\ 3\ \langle\rangle)\\
&= \mathit{baliL}\ (R\ \langle\rangle\ 0\ (R\ \langle\rangle\ 1\ \langle\rangle))\ 2\ (R\ \langle\rangle\ 3\ \langle\rangle)\\
&= R\ (B\ \langle\rangle\ 0\ \langle\rangle)\ 1\ (B\ \langle\rangle\ 2\ (R\ \langle\rangle\ 3\ \langle\rangle))
\end{aligned}
$$

Passing a red node up means an overflow occurred (as in 2-3 trees) that needs to be dealt with further up. At the latest, *insert* turns red into black at the very top.

Function *ins* preserves *invh* but not *invc*: it may return a tree with a red-red conflict at the root, as in the example above: $\mathit{ins}\ 1\ (R\ \langle\rangle\ 0\ \langle\rangle) = R\ \langle\rangle\ 0\ (R\ \langle\rangle\ 1\ \langle\rangle)$. However, once the root node is colored black, everything is fine again. Thus we introduce the weaker invariant *invc2*:

$$\mathit{invc2}\ t \equiv \mathit{invc}\ (\mathit{paint}\ \mathit{Black}\ t)$$

It is easy to prove that *baliL* and *baliR* preserve *invh* and upgrade from *invc2* to *invc*:

$$\mathit{invh}\ l \wedge \mathit{invh}\ r \wedge \mathit{invc2}\ l \wedge \mathit{invc}\ r \wedge \mathit{bh}\ l = \mathit{bh}\ r \longrightarrow$$
$$\mathit{invc}\ (\mathit{baliL}\ l\ a\ r) \wedge \mathit{invh}\ (\mathit{baliL}\ l\ a\ r) \wedge \mathit{bh}\ (\mathit{baliL}\ l\ a\ r) = \mathit{bh}\ l + 1$$

$$\mathit{invh}\ l \wedge \mathit{invh}\ r \wedge \mathit{invc}\ l \wedge \mathit{invc2}\ r \wedge \mathit{bh}\ l = \mathit{bh}\ r \longrightarrow$$
$$\mathit{invc}\ (\mathit{baliR}\ l\ a\ r) \wedge \mathit{invh}\ (\mathit{baliR}\ l\ a\ r) \wedge \mathit{bh}\ (\mathit{baliR}\ l\ a\ r) = \mathit{bh}\ l + 1$$

Another easy induction yields

$$\mathit{invc}\ t \wedge \mathit{invh}\ t \longrightarrow$$
$$\mathit{invc2}\ (\mathit{ins}\ x\ t) \wedge (\mathit{color}\ t = \mathit{Black} \longrightarrow \mathit{invc}\ (\mathit{ins}\ x\ t)) \wedge$$
$$\mathit{invh}\ (\mathit{ins}\ x\ t) \wedge \mathit{bh}\ (\mathit{ins}\ x\ t) = \mathit{bh}\ t$$

The corollary $\mathit{rbt}\ t \longrightarrow \mathit{rbt}\ (\mathit{insert}\ x\ t)$ is immediate.

8.2.2 Deletion

Deletion from a red-black tree is shown in Figure 8.2. It follows the deletion-by-replacing approach (Section 5.2.1). The tricky bit is how to maintain the invariants. As before, intermediate trees may only satisfy the weaker invariant *invc2*. Functions *del* and *split_min* decrease the black height of a tree with a black root node and leave the black height unchanged otherwise. To see that this makes sense, consider deletion from a singleton black or red node. The case that the element to be removed is not in the black tree can be dealt with by coloring the root node red. These are the precise input/output relations:

Lemma 8.1. $\mathit{split\_min}\ t = (x, t') \wedge t \neq \langle\rangle \wedge \mathit{invh}\ t \wedge \mathit{invc}\ t \longrightarrow$
$\mathit{invh}\ t' \wedge (\mathit{color}\ t = \mathit{Red} \longrightarrow \mathit{bh}\ t' = \mathit{bh}\ t \wedge \mathit{invc}\ t') \wedge$
$(\mathit{color}\ t = \mathit{Black} \longrightarrow \mathit{bh}\ t' = \mathit{bh}\ t - 1 \wedge \mathit{invc2}\ t')$

Lemma 8.2. $\mathit{invh}\ t \wedge \mathit{invc}\ t \wedge t' = \mathit{del}\ x\ t \longrightarrow$
$\mathit{invh}\ t' \wedge (\mathit{color}\ t = \mathit{Red} \longrightarrow \mathit{bh}\ t' = \mathit{bh}\ t \wedge \mathit{invc}\ t') \wedge$
$(\mathit{color}\ t = \mathit{Black} \longrightarrow \mathit{bh}\ t' = \mathit{bh}\ t - 1 \wedge \mathit{invc2}\ t')$

It is easy to see that the *del*-Lemma implies correctness of *delete*:

Corollary 8.3. $\mathit{rbt}\ t \longrightarrow \mathit{rbt}\ (\mathit{delete}\ x\ t)$

```
delete x t = paint Black (del x t)

del :: 'a ⇒ 'a rbt ⇒ 'a rbt
del _ ⟨⟩ = ⟨⟩
del x ⟨l, (a, _), r⟩
= (case cmp x a of
    LT ⇒ let l' = del x l
           in if l ≠ ⟨⟩ ∧ color l = Black then baldL l' a r else R l' a r |
    EQ ⇒ if r = ⟨⟩ then l
           else let (a', r') = split_min r
                in if color r = Black then baldR l a' r' else R l a' r' |
    GT ⇒ let r' = del x r
           in if r ≠ ⟨⟩ ∧ color r = Black then baldR l a r' else R l a r')

split_min :: 'a rbt ⇒ 'a × 'a rbt
split_min ⟨l, (a, _), r⟩
= (if l = ⟨⟩ then (a, r)
    else let (x, l') = split_min l
         in (x, if color l = Black then baldL l' a r else R l' a r))

baldL :: 'a rbt ⇒ 'a ⇒ 'a rbt ⇒ 'a rbt
baldL (R t_1 a t_2) b t_3 = R (B t_1 a t_2) b t_3
baldL t_1 a (B t_2 b t_3) = baliR t_1 a (R t_2 b t_3)
baldL t_1 a (R (B t_2 b t_3) c t_4) = R (B t_1 a t_2) b (baliR t_3 c (paint Red t_4))
baldL t_1 a t_2 = R t_1 a t_2

baldR :: 'a rbt ⇒ 'a ⇒ 'a rbt ⇒ 'a rbt
baldR t_1 a (R t_2 b t_3) = R t_1 a (B t_2 b t_3)
baldR (B t_1 a t_2) b t_3 = baliL (R t_1 a t_2) b t_3
baldR (R t_1 a (B t_2 b t_3)) c t_4 = R (baliL (paint Red t_1) a t_2) b (B t_3 c t_4)
baldR t_1 a t_2 = R t_1 a t_2
```

Figure 8.2 Deletion from red-black tree

The proofs of the two preceding lemmas need the following precise characterizations of *baldL* and *baldR*, the counterparts of *baliL* and *baliR*:

Lemma 8.4.
$\textit{invh}\ l \wedge \textit{invh}\ r \wedge \textit{bh}\ l + 1 = \textit{bh}\ r \wedge \textit{invc2}\ l \wedge \textit{invc}\ r \wedge t' = \textit{baldL}\ l\ a\ r \longrightarrow$
$\textit{invh}\ t' \wedge \textit{bh}\ t' = \textit{bh}\ r \wedge \textit{invc2}\ t' \wedge (\textit{color}\ r = \textit{Black} \longrightarrow \textit{invc}\ t')$

Lemma 8.5.
$\textit{invh}\ l \wedge \textit{invh}\ r \wedge \textit{bh}\ l = \textit{bh}\ r + 1 \wedge \textit{invc}\ l \wedge \textit{invc2}\ r \wedge t' = \textit{baldR}\ l\ a\ r \longrightarrow$
$\textit{invh}\ t' \wedge \textit{bh}\ t' = \textit{bh}\ l \wedge \textit{invc2}\ t' \wedge (\textit{color}\ l = \textit{Black} \longrightarrow \textit{invc}\ t')$

The proofs of the two preceding lemmas are by case analyses over the defining equations using the characteristic properties of *baliL* and *baliR* given above.

Proof. Lemma 8.2 is proved by induction on the computation of *del* $x\ t$. The base case is trivial. In the induction step $t = \langle l, (a, c), r\rangle$. If $x < a$ then we distinguish three subcases. If $l = \langle\rangle$ the claim is trivial. Otherwise the claim follows from the IH: if $\textit{color}\ l = \textit{Red}$ then the claim follows directly, if $\textit{color}\ l = \textit{Black}$ then it follows with the help of Lemma 8.4 (with $l = \textit{del}\ x\ l$). The case $a < x$ is dual and the case $x = a$ is similar (using Lemma 8.1). We do not show the details because they are tedious but routine. □

The proof of Lemma 8.1 is similar but simpler.

8.2.3 Deletion by Joining

As an alternative to deletion by replacement we also consider deletion by joining (see Section 5.2.1). The code for red-black trees is shown in Figure 8.3: compared to Figure 8.2, the *EQ* case of *del* has changed and *join* is new.

Invariant preservation is proved much like before except that instead of *split_min* we now have *join* to take care of. The characteristic lemma is proved by induction on the computation of *join*:

Lemma 8.6. $\textit{invh}\ l \wedge \textit{invh}\ r \wedge \textit{bh}\ l = \textit{bh}\ r \wedge \textit{invc}\ l \wedge \textit{invc}\ r \wedge t' = \textit{join}\ l\ r \longrightarrow$
$\textit{invh}\ t' \wedge \textit{bh}\ t' = \textit{bh}\ l \wedge \textit{invc2}\ t' \wedge$
$(\textit{color}\ l = \textit{Black} \wedge \textit{color}\ r = \textit{Black} \longrightarrow \textit{invc}\ t')$

8.3 Implementation of ADT *Map*

Maps based on red-black trees are of course very similar to the above sets. In particular we can reuse the balancing and other auxiliary functions because they do not examine the contents of the nodes but only the color. We follow the general approach in Section 6.5. The representing type is $('a \times 'b)\ \textit{rbt}$.

```
del :: 'a ⇒ 'a rbt ⇒ 'a rbt
del _ ⟨⟩ = ⟨⟩
del x ⟨l, (a, _), r⟩
= (case cmp x a of
    LT ⇒ if l ≠ ⟨⟩ ∧ color l = Black then baldL (del x l) a r
           else R (del x l) a r |
    EQ ⇒ join l r |
    GT ⇒ if r ≠ ⟨⟩ ∧ color r = Black then baldR l a (del x r)
           else R l a (del x r))

join :: 'a rbt ⇒ 'a rbt ⇒ 'a rbt
join ⟨⟩ t = t
join t ⟨⟩ = t
join (R t1 a t2) (R t3 c t4)
= (case join t2 t3 of
    R u2 b u3 ⇒ R (R t1 a u2) b (R u3 c t4) |
    t23 ⇒ R t1 a (R t23 c t4))
join (B t1 a t2) (B t3 c t4)
= (case join t2 t3 of
    R u2 b u3 ⇒ R (B t1 a u2) b (B u3 c t4) |
    t23 ⇒ baldL t1 a (B t23 c t4))
join t1 (R t2 a t3) = R (join t1 t2) a t3 |
join (R t1 a t2) t3 = R t1 a (join t2 t3)
```

Figure 8.3 Deletion from red-black tree by joining

Function *lookup* is almost identical to its precursor in Section 6.5 except that the lhs of the recursive case is *lookup* $\langle l, ((a, b), \_), r\rangle\ x$ because of the (irrelevant) color field. There is no need to show the code.

Function *update* is shown in Figure 8.4. It is a minor variation of insertion shown in Figure 8.1.

Deletion can be implemented by replacing and by joining. (In the source files we have chosen the second option.) In both cases, all we need is to adapt *del* for sets by replacing *cmp* $x\ a$ by *cmp* x (*fst* a) (where the second a is of type $'a \times 'b$ and should be renamed, e.g. to ab). Again, there is no need to show the code.

```
update :: 'a ⇒ 'b ⇒ ('a × 'b) rbt ⇒ ('a × 'b) rbt
update x y t = paint Black (upd x y t)

upd :: 'a ⇒ 'b ⇒ ('a × 'b) rbt ⇒ ('a × 'b) rbt
upd x y ⟨⟩ = R ⟨⟩ (x, y) ⟨⟩
upd x y (B l (a, b) r) = (case cmp x a of
                            LT ⇒ baliL (upd x y l) (a, b) r |
                            EQ ⇒ B l (x, y) r |
                            GT ⇒ baliR l (a, b) (upd x y r))
upd x y (R l (a, b) r) = (case cmp x a of
                            LT ⇒ R (upd x y l) (a, b) r |
                            EQ ⇒ R l (x, y) r |
                            GT ⇒ R l (a, b) (upd x y r))
```

Figure 8.4 Red-black tree map update

8.4 Exercises

Exercise 8.1. Show that the logarithmic height of red-black trees is already guaranteed by the color and height invariants:

$$\mathit{invc}\ t \wedge \mathit{invh}\ t \longrightarrow h\ t \le 2 \cdot \lg |t|_1 + 2$$

Exercise 8.2. We already discussed informally why the definition of *invh* captures "all paths from the root to a leaf have the same number of black nodes" although *bh* only traverses the left spine. This exercise formalizes that discussion. The following function computes the set of black heights (number of black nodes) of all paths:

$$\mathit{bhs} :: {}'a\ \mathit{rbt} \Rightarrow \mathit{nat}\ \mathit{set}$$

$$\mathit{bhs}\ \langle\rangle = \{0\}$$
$$\mathit{bhs}\ \langle l, (\_, c), r\rangle$$
$$= (\mathbf{let}\ H = \mathit{bhs}\ l \cup \mathit{bhs}\ r\ \mathbf{in}\ \mathbf{if}\ c = \mathit{Black}\ \mathbf{then}\ \mathit{Suc}\ \text{`}\ H\ \mathbf{else}\ H)$$

where the infix operator (`) is predefined as $f\ \text{`}\ A = \{y \mid \exists x{\in}A.\ y = f\ x\}$. Prove $\mathit{invh}\ t \longleftrightarrow \mathit{bhs}\ t = \{\mathit{bh}\ t\}$. The $\longrightarrow$ direction should be easy, the other direction should need some lemmas.

Exercise 8.3. Following Section 7.3, define a linear-time function $\mathit{rbt\_of\_list} :: {}'a\ \mathit{list} \Rightarrow {}'a\ \mathit{rbt}$ and prove $\mathit{inorder}\ (\mathit{rbt\_of\_list}\ \mathit{as}) = \mathit{as}$ and $\mathit{rbt}\ (\mathit{rbt\_of\_list}\ \mathit{as})$.

Chapter Notes

Red-black trees were invented by Bayer [1972] who called them “symmetric binary B-trees”. The red-black color convention was introduced by Guibas and Sedgewick [1978] who studied their properties in greater depth. The first functional version of red-black trees (without deletion) is due to Okasaki [1998] and everybody follows his code. A functional version of deletion was first given by Kahrs [2001] and Section 8.2.3 is based on it. Germane and Might [2014] presents a function for deletion by replacement that is quite different from the one in Section 8.2.2. Our starting point was an Isabelle proof by Reiter and Krauss (based on Kahrs). Other verifications of red-black trees are reported by Filliâtre and Letouzey [2004] (using their own deletion function) and Appel [2011] (based on Kahrs).

9 AVL Trees

Tobias Nipkow

The AVL tree (named after its inventors Adel'son-Vel'skiĭ and Landis [1962]) is the granddaddy of efficient binary search trees. Its logarithmic height is maintained by rotating subtrees based on their height. For efficiency reasons the height of each subtree is stored in its root node. That is, the underlying data structure is a height-augmented tree (see Section 4.4):

type_synonym $'a\ tree\_ht = ('a \times nat)\ tree$

Function *ht* extracts the height field and *node* is a smart constructor that sets the height field:

$\mathit{ht} :: 'a\ tree\_ht \Rightarrow nat$

$\mathit{ht}\ \langle\rangle = 0$

$\mathit{ht}\ \langle\_,\ (\_,\ n),\ \_\rangle = n$

$\mathit{node} :: 'a\ tree\_ht \Rightarrow 'a \Rightarrow 'a\ tree\_ht \Rightarrow 'a\ tree\_ht$

$\mathit{node}\ l\ a\ r = \langle l,\ (a,\ \mathit{max}\ (\mathit{ht}\ l)\ (\mathit{ht}\ r) + 1),\ r\rangle$

An **AVL tree** is a tree that satisfies the AVL invariant: the height of the left and right child of any node differ by at most 1

$\mathit{avl} :: 'a\ tree\_ht \Rightarrow bool$

$\mathit{avl}\ \langle\rangle = \mathit{True}$

$\mathit{avl}\ \langle l,\ (\_,\ n),\ r\rangle$

$= (|\mathit{int}\ (\mathit{h}\ l) - \mathit{int}\ (\mathit{h}\ r)| \le 1 \land n = \mathit{max}\ (\mathit{h}\ l)\ (\mathit{h}\ r) + 1 \land \mathit{avl}\ l \land \mathit{avl}\ r)$

and the height field contains the correct value. The conversion function $\mathit{int} :: nat \Rightarrow int$ is required because on natural numbers $0 - n = 0$.

9.1 Logarithmic Height

AVL trees have logarithmic height. The key insight for the proof is that $M\ n$, the minimal number of leaves of an AVL tree of height n, satisfies the recurrence relation $M\ (n + 2) = M\ (n + 1) + M\ n$. Instead of formalizing this function M we prove directly that an AVL tree of height n has at least $\mathit{fib}\ (n + 2)$ leaves where fib is the Fibonacci function:

$$\mathit{fib} :: \mathit{nat} \Rightarrow \mathit{nat}$$

$$\begin{aligned}
&\mathit{fib}\ 0 = 0\\
&\mathit{fib}\ 1 = 1\\
&\mathit{fib}\ (n + 2) = \mathit{fib}\ (n + 1) + \mathit{fib}\ n
\end{aligned}$$

Lemma 9.1. $\mathit{avl}\ t \longrightarrow \mathit{fib}\ (h\ t + 2) \le |t|_1$

Proof. The proof is by induction on t. We focus on the induction step $t = \langle l, (a, n), r\rangle$ and assume $\mathit{avl}\ t$. Thus the IHs reduce to $\mathit{fib}\ (h\ l + 2) \le |l|_1$ and $\mathit{fib}\ (h\ r + 2) \le |r|_1$. We prove $\mathit{fib}\ (\mathit{max}\ (h\ l)\ (h\ r) + 3) \le |l|_1 + |r|_1$, from which $\mathit{avl}\ t \longrightarrow \mathit{fib}\ (h\ t + 2) \le |t|_1$ follows directly. There are two cases. We focus on $h\ l \ge h\ r$, $h\ l < h\ r$ is dual.

$$\begin{aligned}
&\mathit{fib}\ (\mathit{max}\ (h\ l)\ (h\ r) + 3) = \mathit{fib}\ (h\ l + 3)\\
&= \mathit{fib}\ (h\ l + 2) + \mathit{fib}\ (h\ l + 1)\\
&\le |l|_1 + \mathit{fib}\ (h\ l + 1) && \text{by } \mathit{fib}\ (h\ l + 2) \le |l|_1\\
&\le |l|_1 + |r|_1 && \text{by } \mathit{fib}\ (h\ r + 2) \le |r|_1
\end{aligned}$$

The last step is justified because $h\ l + 1 \le h\ r + 2$ (which follows from $\mathit{avl}\ t$) and fib is monotone. □

Now we prove a well-known exponential lower bound for fib where $\varphi \equiv (1 + \sqrt{5}) / 2$:

Lemma 9.2. $\varphi^n \le \mathit{fib}\ (n + 2)$

Proof. The proof is by induction on n by fib computation induction. The case $n = 0$ is trivial and the case $n = 1$ is easy. Now consider the induction step:

$$\begin{aligned}
&\mathit{fib}\ (n + 2 + 2) = \mathit{fib}\ (n + 2 + 1) + \mathit{fib}\ (n + 2)\\
&\ge \varphi^{n + 1} + \varphi^n && \text{by IHs}\\
&= (\varphi + 1) \cdot \varphi^n\\
&= \varphi^{n + 2} && \text{because } \varphi + 1 = \varphi^2
\end{aligned}$$
□

Combining the two lemmas yields $\mathit{avl}\ t \longrightarrow \varphi^{h\ t} \le |t|_1$ and thus

Corollary 9.3. $\mathit{avl}\ t \longrightarrow h\ t \le 1 / \mathit{lg}\ \varphi \cdot \mathit{lg}\ |t|_1$

That is, the height of an AVL tree is at most $1\ /\ \mathit{lg}\ \varphi \approx 1.44$ times worse than the optimal $\mathit{lg}\ |t|_1$.

9.2 Implementation of ADT *Set*

9.2.1 Insertion

Insertion follows the standard approach: insert the element as usual and reestablish the AVL invariant on the way back up.

```
insert :: 'a ⇒ 'a tree_ht ⇒ 'a tree_ht
insert x ⟨⟩ = ⟨⟨⟩, (x, 1), ⟨⟩⟩
insert x ⟨l, (a, n), r⟩ = (case cmp x a of
                           LT ⇒ balL (insert x l) a r |
                           EQ ⇒ ⟨l, (a, n), r⟩ |
                           GT ⇒ balR l a (insert x r))
```

Functions *balL*/*balR* readjust the tree after an insertion into the left/right child. The AVL invariant has been lost if the difference in height has become 2 — it cannot become more because the height can only increase by 1. Consider the definition of *balL* in Figure 9.1 (*balR* in Figure 9.2 is dual). If the AVL invariant has not been lost, i.e. if $\mathit{ht}\ AB \neq \mathit{ht}\ C + 2$, then we can just return the AVL tree $\mathit{node}\ AB\ c\ C$. But if $\mathit{ht}\ AB = \mathit{ht}\ C + 2$, we need to “rotate” the subtrees suitably. Clearly AB must be of the form $\langle A,\ (a,\ \_),\ B\rangle$. There are two cases, which are illustrated in Figure 9.1. Triangles of the same height denote trees of the same height. A +1 at the bottom denotes an additional level due to insertion of the new element.

If $\mathit{ht}\ B \leq \mathit{ht}\ A$ then *balL* performs what is known as a single rotation.

If $\mathit{ht}\ A < \mathit{ht}\ B$ then B must be of the form $\langle B_1,\ (b,\ \_),\ B_2\rangle$ (where either B_1 or B_2 has increased in height) and *balL* performs what is known as a double rotation.

It is easy to check that in both cases the tree on the right satisfies the AVL invariant.

Preservation of *avl* by *insert* cannot be proved in isolation but needs to be proved simultaneously with how *insert* changes the height (because *avl* depends on the height and *insert* requires *avl* for correct behaviour):

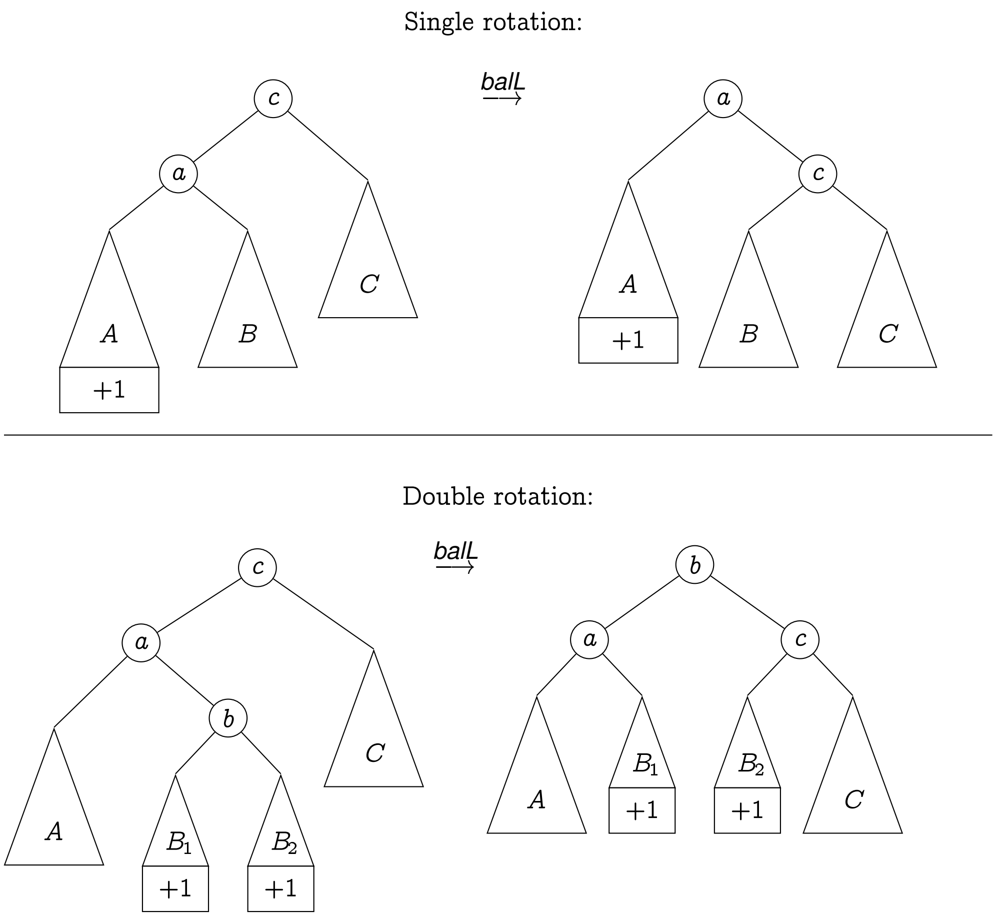
Theorem 9.4. $\mathit{avl}\ t \longrightarrow \mathit{avl}\ (\mathit{insert}\ x\ t) \wedge h\ (\mathit{insert}\ x\ t) \in \{h\ t,\ h\ t + 1\}$

The proof is by induction on t followed by a complete case analysis (which Isabelle automates).

9.2.2 Deletion

Figure 9.3 shows deletion-by-replacing (see Section 5.2.1). The recursive calls are dual to insertion: in terms of the difference in height, deletion of some element from one

```
balL :: 'a tree_ht ⇒ 'a ⇒ 'a tree_ht ⇒ 'a tree_ht
balL AB c C
= (if ht AB = ht C + 2
   then case AB of
          ⟨A, (a, x), B⟩ ⇒
            if ht B ≤ ht A then node A a (node B c C)
            else case B of
                   ⟨B₁, (b, _), B₂⟩ ⇒ node (node A a B₁) b (node B₂ c C)
   else node AB c C)
```


Single rotation:

Double rotation:

Figure 9.1 Function *balL*

```
balR :: 'a tree_ht ⇒ 'a ⇒ 'a tree_ht ⇒ 'a tree_ht
balR A a BC
= (if ht BC = ht A + 2
   then case BC of
          ⟨B, (c, x), C⟩ ⇒
            if ht B ≤ ht C then node (node A a B) c C
            else case B of
                   ⟨B₁, (b, _), B₂⟩ ⇒ node (node A a B₁) b (node B₂ c C)
   else node A a BC)
```

Figure 9.2 Function *balR*

```
delete :: 'a ⇒ 'a tree_ht ⇒ 'a tree_ht
delete _ ⟨⟩ = ⟨⟩
delete x ⟨l, (a, _), r⟩
= (case cmp x a of
   LT ⇒ balR (delete x l) a r |
   EQ ⇒ if l = ⟨⟩ then r else let (l', a') = split_max l in balR l' a' r |
   GT ⇒ balL l a (delete x r))

split_max :: 'a tree_ht ⇒ 'a tree_ht × 'a
split_max ⟨l, (a, _), r⟩
= (if r = ⟨⟩ then (l, a)
   else let (r', a') = split_max r in (balL l a r', a'))
```

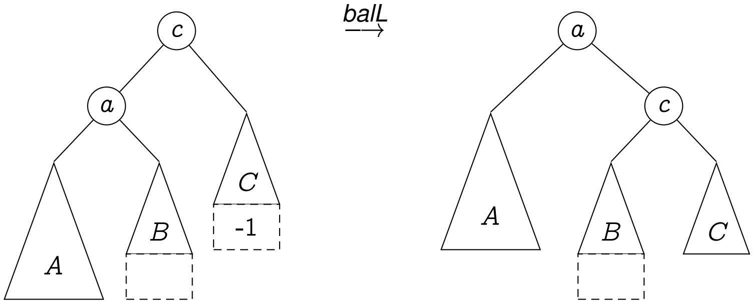
Figure 9.3 Deletion from AVL tree

child is the same as insertion of some element into the other child. Thus functions *balR*/*balL* can again be employed to restore the invariant.

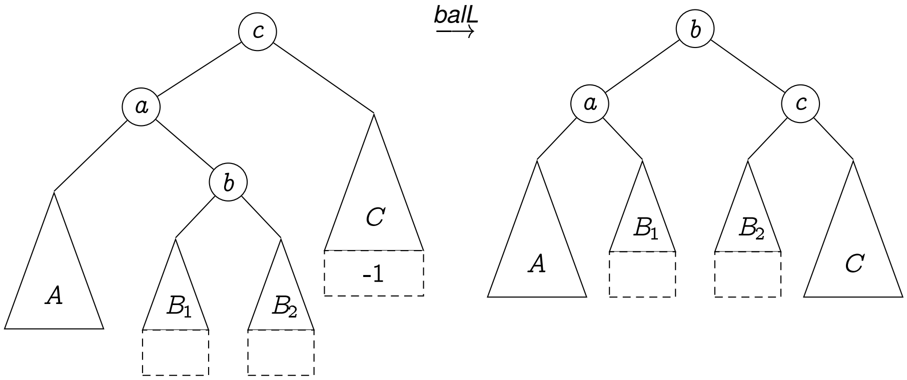
An element is deleted from a node by replacing it with the maximal element of the left child (the minimal element of the right child would work just as well). Function *split_max* performs that extraction and uses *balL* to restore the invariant after splitting an element off the right child.

The fact that *balR*/*balL* can be reused for deletion can be illustrated by drawing the corresponding rotation diagrams. We look at how the code for *balL* behaves when an element has been deleted from C. Dashed rectangles at the bottom indicate a single additional level that may or may not be there. A -1 indicates that the level has disappeared due to deletion.

Single rotation in *balL* after deletion in C:



Double rotation in *balL* after deletion in C:



At least one of B_1 and B_2 must have the same height as A.

Preservation of *avl* by *delete* can be proved in the same manner as for *insert* but we provide more of the details (partly because our Isabelle proof is less automatic). The following lemmas express that the auxiliary functions preserve *avl*:

$$\mathit{avl}\ l \wedge \mathit{avl}\ r \wedge h\ r - 1 \le h\ l \wedge h\ l \le h\ r + 2 \longrightarrow \mathit{avl}\ (\mathit{balL}\ l\ a\ r)$$

$$\mathit{avl}\ l \wedge \mathit{avl}\ r \wedge h\ l - 1 \le h\ r \wedge h\ r \le h\ l + 2 \longrightarrow \mathit{avl}\ (\mathit{balR}\ l\ a\ r)$$

$$\begin{array}{l} \mathit{avl}\ t \wedge t \neq \langle\rangle \longrightarrow \\ \mathit{avl}\ (\mathit{fst}\ (\mathit{split\_max}\ t)) \wedge \\ h\ t \in \{h\ (\mathit{fst}\ (\mathit{split\_max}\ t)),\ h\ (\mathit{fst}\ (\mathit{split\_max}\ t)) + 1\} \end{array}$$

The first two are proved by the obvious cases analyses, the last one also requires induction.

As for *insert*, preservation of *avl* by *delete* needs to be proved simultaneously with how *delete* changes the height:

Theorem 9.5. $\mathit{avl}\ t \wedge t' = \mathit{delete}\ x\ t \longrightarrow \mathit{avl}\ t' \wedge h\ t \in \{h\ t', h\ t' + 1\}$

Proof. The proof is by induction on t followed by the case analyses dictated by the code for *delete*. We sketch the induction step. Let $t = \langle l, (a, n), r\rangle$ and $t' = \mathit{delete}\ x\ t$ and assume the IHs and *avl* t. The claim *avl* t' follows from the preservation of *avl* by *balL*, *balR* and *split_max* as shown above. The claim $h\ t \in \{h\ t', h\ t' + 1\}$ follows directly from the definitions of *balL* and *balR*. □

9.3 Exercises

Exercise 9.1. The logarithmic height of AVL trees can be proved directly. Prove

$$\mathit{avl}\ t \wedge h\ t = n \longrightarrow 2^{n\ \mathrm{div}\ 2} \leq |t|_1$$

by *fib* computation induction on n. This implies $\mathit{avl}\ t \longrightarrow h\ t \leq 2 \cdot \mathit{lg}\ |t|_1$.

Exercise 9.2. **Fibonacci trees** are defined in analogy to Fibonacci numbers:

$$\mathit{fibt} :: \mathit{nat} \Rightarrow \mathit{unit\ tree}$$

$$\mathit{fibt}\ 0 = \langle\rangle$$
$$\mathit{fibt}\ 1 = \langle\langle\rangle, (), \langle\rangle\rangle$$
$$\mathit{fibt}\ (n + 2) = \langle \mathit{fibt}\ (n + 1), (), \mathit{fibt}\ n\rangle$$

We are only interested in the shape of these trees. Therefore the nodes just contain dummy *unit* values (). Hence we need to define the AVL invariant for trees without annotations:

$$\mathit{avl0} :: {}'a\ \mathit{tree} \Rightarrow \mathit{bool}$$

$$\mathit{avl0}\ \langle\rangle = \mathit{True}$$
$$\mathit{avl0}\ \langle l, \_, r\rangle = (|\mathit{int}\ (h\ l) - \mathit{int}\ (h\ r)| \leq 1 \wedge \mathit{avl0}\ l \wedge \mathit{avl0}\ r)$$

Prove the following properties of Fibonacci trees:

$$\mathit{avl0}\ (\mathit{fibt}\ n) \qquad |\mathit{fibt}\ n|_1 = \mathit{fib}\ (n + 2)$$

Conclude that the Fibonacci trees are minimal (w.r.t. their size) among all AVL trees of a given height:

$$\mathit{avl}\ t \longrightarrow |\mathit{fibt}\ (h\ t)|_1 \leq |t|_1$$

Exercise 9.3. Show that every almost complete tree is an AVL tree:

$$\mathit{acomplete}\ t \longrightarrow \mathit{avl0}\ t$$

As in the previous exercise we consider trees without height annotations.

Exercise 9.4. Generalize AVL trees to **height-balanced trees** where the condition

$$|\mathit{int}\ (h\ l) - \mathit{int}\ (h\ r)| \le 1$$

in the invariant is replaced by

$$|\mathit{int}\ (h\ l) - \mathit{int}\ (h\ r)| \le m$$

where $m \ge 1$ is some fixed integer. Modify the invariant and the insertion and deletion functions and prove that the latter fulfill the same correctness theorems as before. You do not need to prove the logarithmic height of height-balanced trees.

Exercise 9.5. Following Section 7.3, define a linear-time function *avl_of_list* :: *'a list* ⇒ *'a tree_ht* and prove both *inorder* (*avl_of_list as*) = *as* and *avl* (*avl_of_list as*).

9.4 An Optimization

Instead of recording the height of the tree in each node, it suffices to record the **balance factor**, i.e. the difference in height of its two children. Rather than the three integers -1, 0 and 1 we utilize a new data type:

```
datatype bal = Lh | Bal | Rh

type_synonym 'a tree_bal = ('a × bal) tree
```

The names *Lh* and *Rh* stand for “left-heavy” and “right-heavy”. The AVL invariant for these trees reflect these names:

```
avl :: 'a tree_bal ⇒ bool
avl ⟨⟩ = True
avl ⟨l, (_, b), r⟩ = ((case b of
                         Lh ⇒ h l = h r + 1 |
                         Bal ⇒ h r = h l |
                         Rh ⇒ h r = h l + 1) ∧
                        avl l ∧ avl r)
```

The code for insertion (and deletion) is similar to the height-based version. The key difference is that the test if the AVL invariant has been lost cannot be based on the height anymore. We need to detect if the tree has increased in height upon insertion based on the balance factors. The key insight is that a height increase is coupled with a change from *Bal* to *Lh* or *Rh*, except when we transition from ⟨⟩ to ⟨⟨⟩, (*a*, *Bal*), ⟨⟩⟩. This insight is encoded in the test *incr*:

```
is_bal :: 'a tree_bal ⇒ bool
is_bal ⟨_, (_, b), _⟩ = (b = Bal)

incr :: 'a tree_bal ⇒ 'b tree_bal ⇒ bool
incr t t' = (t = ⟨⟩ ∨ is_bal t ∧ ¬ is_bal t')
```

The test for a height increase compares the trees before and after insertion. Therefore it has been pulled out of the balance functions into insertion:

```
insert :: 'a ⇒ 'a tree_bal ⇒ 'a tree_bal
insert x ⟨⟩ = ⟨⟨⟩, (x, Bal), ⟨⟩⟩
insert x ⟨l, (a, b), r⟩
= (case cmp x a of
     LT ⇒ let l' = insert x l
           in if incr l l' then balL l' a b r else ⟨l', (a, b), r⟩ |
     EQ ⇒ ⟨l, (a, b), r⟩ |
     GT ⇒ let r' = insert x r
            in if incr r r' then balR l a b r' else ⟨l, (a, b), r'⟩)
```

The balance functions are shown in Figure 9.4. Function *rot2* implements double rotations. Function *balL* is called if the left child AB has increased in height. If the tree was *Lh* then single or double rotations are necessary to restore balance. Otherwise we simply need to adjust the balance factors. Function *balR* is dual to *balL*.

For deletion we need to test if the height has decreased and *decr* implements this test:

```
decr :: 'a tree_bal ⇒ 'b tree_bal ⇒ bool
decr t t' = (t ≠ ⟨⟩ ∧ incr t' t)
```

Function *decr* is almost the dual of *incr* except that *decr* must also ensure $t \neq \langle\rangle$. In places where $t \neq \langle\rangle$ is already guaranteed, we have replaced *decr* t t' by *incr* t' t.

```
balL :: 'a tree_bal ⇒ 'a ⇒ bal ⇒ 'a tree_bal ⇒ 'a tree_bal
balL AB c bc C
= (case bc of
    Lh ⇒ case AB of
           ⟨A, (a, Lh), B⟩ ⇒ ⟨A, (a, Bal), ⟨B, (c, Bal), C⟩⟩ |
           ⟨A, (a, Bal), B⟩ ⇒ ⟨A, (a, Rh), ⟨B, (c, Lh), C⟩⟩ |
           ⟨A, (a, Rh), B⟩ ⇒ rot2 A a B c C |
    Bal ⇒ ⟨AB, (c, Lh), C⟩ |
    Rh ⇒ ⟨AB, (c, Bal), C⟩)

balR :: 'a tree_bal ⇒ 'a ⇒ bal ⇒ 'a tree_bal ⇒ 'a tree_bal
balR A a ba BC
= (case ba of
    Lh ⇒ ⟨A, (a, Bal), BC⟩ |
    Bal ⇒ ⟨A, (a, Rh), BC⟩ |
    Rh ⇒ case BC of
           ⟨B, (c, Lh), C⟩ ⇒ rot2 A a B c C |
           ⟨B, (c, Bal), C⟩ ⇒ ⟨⟨A, (a, Rh), B⟩, (c, Lh), C⟩ |
           ⟨B, (c, Rh), C⟩ ⇒ ⟨⟨A, (a, Bal), B⟩, (c, Bal), C⟩)

rot2 :: 'a tree_bal ⇒ 'a ⇒ 'a tree_bal ⇒ 'a ⇒ 'a tree_bal ⇒ 'a tree_bal
rot2 A a B c C
= (case B of
    ⟨B_1, (b, bb), B_2⟩ ⇒
      let b_1 = if bb = Rh then Lh else Bal;
          b_2 = if bb = Lh then Rh else Bal
      in ⟨⟨A, (a, b_1), B_1⟩, (b, Bal), ⟨B_2, (c, b_2), C⟩⟩)
```

Figure 9.4 Functions *balL* and *balR*

Deletion and *split_max* change in the same manner as insertion:

```
delete :: 'a ⇒ 'a tree_bal ⇒ 'a tree_bal
delete _ ⟨⟩ = ⟨⟩
delete x ⟨l, (a, ba), r⟩
= (case cmp x a of
    LT ⇒ let l' = delete x l
          in if decr l l' then balR l' a ba r else ⟨l', (a, ba), r⟩
    | EQ ⇒ if l = ⟨⟩ then r
            else let (l', a') = split_max l
                 in if incr l' l then balR l' a' ba r
                    else ⟨l', (a', ba), r⟩
    | GT ⇒ let r' = delete x r
            in if decr r r' then balL l a ba r' else ⟨l, (a, ba), r'⟩)

split_max :: 'a tree_bal ⇒ 'a tree_bal × 'a
split_max ⟨l, (a, ba), r⟩
= (if r = ⟨⟩ then (l, a)
   else let (r', a') = split_max r;
            t' = if incr r' r then balL l a ba r' else ⟨l, (a, ba), r'⟩
        in (t', a'))
```

In the end we have the following correctness theorems:

Theorem 9.6. $\mathit{avl}\ t \wedge t' = \mathit{insert}\ x\ t \longrightarrow$
$\mathit{avl}\ t' \wedge h\ t' = h\ t + (\textbf{if}\ \mathit{incr}\ t\ t'\ \textbf{then}\ 1\ \textbf{else}\ 0)$

This theorem tells us not only that *avl* is preserved but also that *incr* indicates correctly if the height has increased or not. Similarly for deletion and *decr*:

Theorem 9.7. $\mathit{avl}\ t \wedge t' = \mathit{delete}\ x\ t \longrightarrow$
$\mathit{avl}\ t' \wedge h\ t = h\ t' + (\textbf{if}\ \mathit{decr}\ t\ t'\ \textbf{then}\ 1\ \textbf{else}\ 0)$

The proofs of both theorems follow the standard pattern of induction followed by an exhaustive (automatic) cases analysis. The proof for *delete* requires an analogous lemma for *split_max*:

$$\mathit{split\_max}\ t = (t', a) \wedge \mathit{avl}\ t \wedge t \neq \langle\rangle \longrightarrow$$
$$\mathit{avl}\ t' \wedge h\ t = h\ t' + (\textbf{if}\ \mathit{incr}\ t'\ t\ \textbf{then}\ 1\ \textbf{else}\ 0)$$

9.5 Exercises

Exercise 9.6. We map type $'a\ tree\_bal$ back to type $('a \times nat)\ tree$ (called $'a\ tree\_ht$ in the beginning of the chapter):

$$\mathit{debal} :: 'a\ tree\_bal \Rightarrow ('a \times nat)\ tree$$

$$\mathit{debal}\ \langle\rangle = \langle\rangle$$
$$\mathit{debal}\ \langle l,\ (a,\ \_),\ r\rangle = \langle \mathit{debal}\ l,\ (a,\ \mathit{max}\ (h\ l)\ (h\ r) + 1),\ \mathit{debal}\ r\rangle$$

Prove that the AVL property is preserved: $\mathit{avl}\ t \longrightarrow \mathit{avl\_ht}\ (\mathit{debal}\ t)$ where $\mathit{avl\_ht}$ is defined in the beginning of the chapter.

Define a function $\mathit{debal2}$ of the same type that traverses the tree only once and in particular does not use function h. Prove $\mathit{avl}\ t \longrightarrow \mathit{debal2}\ t = \mathit{debal}\ t$.

Exercise 9.7. To realize the full space savings potential of balance factors we encode them directly into the node constructors and work with the following special tree type:

```
datatype 'a tree4 = Leaf
  | Lh ('a tree4) 'a ('a tree4)
  | Bal ('a tree4) 'a ('a tree4)
  | Rh ('a tree4) 'a ('a tree4)
```

On this type, define the AVL invariant, insertion, deletion and all necessary auxiliary functions. Prove theorems 9.6 and 9.7. Hint: modify the theory underlying Section 9.4.

10 Beyond Insert and Delete: $\cup$, $\cap$ and $-$

Tobias Nipkow

So far we looked almost exclusively at insertion and deletion of single elements, with the exception of the conversion of whole lists of elements into search trees. This chapter is dedicated to operations that combine two sets (implemented by search trees) by union, intersection and difference. We denote set difference by $-$ rather than $\setminus$.

Let us focus on set union for a moment and assume that insertion into a set of size s takes time proportional to $\lg s$. Consider two sets A and B of size m and n where $m \leq n$. The naive approach is to insert the elements from one set one by one into the other set. This takes time proportional to $\lg n + \cdots + \lg(n+m-1)$ or $\lg m + \cdots + \lg(m+n-1)$ depending on whether the smaller set is inserted into the larger one or the other way around. Of course the former sum is less than or equal to the latter sum. To estimate the growth of $\lg n + \cdots + \lg(n+m-1) = \lg(n \cdots (n+m-1))$ we can easily generalize the derivation of $\lg(n!) \in \Theta(n \lg n)$ in the initial paragraph of Section 7.3. The result is $\lg(n \cdots (n+m-1)) \in \Theta(m \lg n)$. That is, inserting m elements into an n element set one by one takes time $\Theta(m \lg n)$.

There is a second, possibly naive sounding algorithm for computing the union: flatten both trees to ordered lists (using function *inorder2* from Exercise 4.1), merge both lists and convert the resulting list back into a suitably balanced search tree. All three steps take linear time. The last step is the only slightly nontrivial one but has been dealt with before (see Section 7.3 and Exercises 8.3 and 9.5). This algorithm takes time $O(m+n)$ which is significantly better than $O(m \lg n)$ if $m \approx n$ but significantly worse if $m << n$.

This chapter presents a third approach that has the following salient features:

- Union, intersection and difference take time $O(m \lg(\frac{n}{m}+1))$
- It works for a whole class of balanced trees, including AVL, red-black and weight-balanced trees.
- It is based on a single function for joining two balanced trees to form a new balanced tree.

ADT $\mathit{Set2} = \mathit{Set} +$

interface

$\mathit{union} :: {'s} \Rightarrow {'s} \Rightarrow {'s}$
$\mathit{inter} :: {'s} \Rightarrow {'s} \Rightarrow {'s}$
$\mathit{diff} :: {'s} \Rightarrow {'s} \Rightarrow {'s}$

specification

$\mathit{invar}\ s_1 \wedge \mathit{invar}\ s_2 \longrightarrow \mathit{set}\ (\mathit{union}\ s_1\ s_2) = \mathit{set}\ s_1 \cup \mathit{set}\ s_2$ (*union*)
$\mathit{invar}\ s_1 \wedge \mathit{invar}\ s_2 \longrightarrow \mathit{invar}\ (\mathit{union}\ s_1\ s_2)$ (*union-inv*)
$\mathit{invar}\ s_1 \wedge \mathit{invar}\ s_2 \longrightarrow \mathit{set}\ (\mathit{inter}\ s_1\ s_2) = \mathit{set}\ s_1 \cap \mathit{set}\ s_2$ (*inter*)
$\mathit{invar}\ s_1 \wedge \mathit{invar}\ s_2 \longrightarrow \mathit{invar}\ (\mathit{inter}\ s_1\ s_2)$ (*inter-inv*)
$\mathit{invar}\ s_1 \wedge \mathit{invar}\ s_2 \longrightarrow \mathit{set}\ (\mathit{diff}\ s_1\ s_2) = \mathit{set}\ s_1 - \mathit{set}\ s_2$ (*diff*)
$\mathit{invar}\ s_1 \wedge \mathit{invar}\ s_2 \longrightarrow \mathit{invar}\ (\mathit{diff}\ s_1\ s_2)$ (*diff-inv*)

Figure 10.1 ADT *Set2*

We call it the **join approach**. It is easily and efficiently parallelizable, a property we will not explore here.

The join approach is at least as fast as the one-by-one approach: from $m+n \leq mn$ it follows that $\frac{n}{m}+1 \leq n$ (if $m,n \geq 2$). The join approach is also at least as fast as the tree-to-list-to-tree approach because $m+n = m(\frac{n}{m}+1)$ (if $m \geq 1$).

10.1 Specification of Union, Intersection and Difference

Before explaining the join approach we extend the ADT *Set* by three new functions *union*, *inter* and *diff*. The specification in Figure 10.1 is self-explanatory.

10.2 Just Join

Now we come to the heart of the matter, the definition of union, intersection and difference in terms of a single function *join*. We promised that the algorithms would be generic across a range of balanced trees. Thus we assume that we operate on augmented trees of type $({'a} \times {'b})\ \mathit{tree}$ where ${'a}$ is the type of the elements and ${'b}$ is the balancing information (which we can ignore here). This enables us to formulate the algorithms via pattern-matching. A more generic approach is the subject of Exercise 10.2.

The whole section is parameterized by the join function and an invariant:

$$\mathit{join} :: ({'a} \times {'b})\ \mathit{tree} \Rightarrow {'a} \Rightarrow ({'a} \times {'b})\ \mathit{tree} \Rightarrow ({'a} \times {'b})\ \mathit{tree}$$
$$\mathit{inv} :: ({'a} \times {'b})\ \mathit{tree} \Rightarrow \mathit{bool}$$

$$\textsf{set\_tree}\ (join\ l\ a\ r) = \textsf{set\_tree}\ l \cup \{a\} \cup \textsf{set\_tree}\ r \tag{10.1}$$

$$\textsf{bst}\ \langle l,\ (a,\ \_),\ r\rangle \longrightarrow \textsf{bst}\ (join\ l\ a\ r) \tag{10.2}$$

$$inv\ \langle\rangle$$

$$inv\ l \wedge inv\ r \longrightarrow inv\ (join\ l\ a\ r) \tag{10.3}$$

$$inv\ \langle l,\ (\_,\ \_),\ r\rangle \longrightarrow inv\ l \wedge inv\ r \tag{10.4}$$

Figure 10.2 Specification of *join* and *inv*

Function *inv* is meant to take care of the balancedness property only, not the BST property. Functions *join* and *inv* are specified with the help of the standard tree functions *set_tree* and *bst* in Figure 10.2. With respect to the set of elements, *join* must behave like union. But it need only return a BST if both trees are BSTs and the element a lies in between the elements of the two trees, i.e. if *bst* $\langle l,\ (a,\ \_),\ r\rangle$. The structural invariant *inv* must be preserved by formation and destruction of trees. Thus we can see *join* as a smart constructor that builds a balanced tree.

To define union and friends we need a number of simple auxiliary functions. Function *split_min* decomposes a tree into its leftmost (minimal) element and the remaining tree; the remaining tree is reassembled via *join*, thus preserving *inv*:

```
split_min :: ('a × 'b) tree ⇒ 'a × ('a × 'b) tree
split_min ⟨l, (a, _), r⟩
= (if l = ⟨⟩ then (a, r)
   else let (m, l') = split_min l in (m, join l' a r))
```

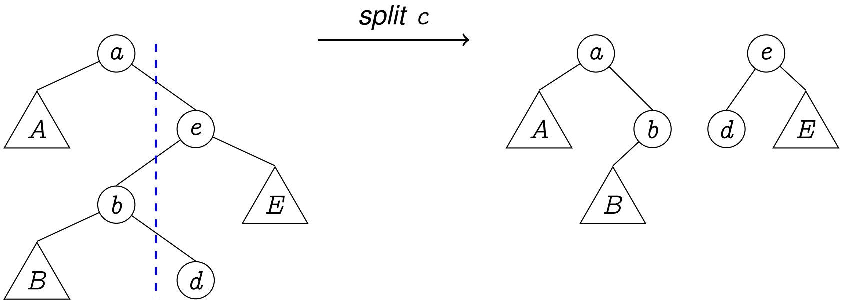
Function *join2* is reduced to *join* with the help of *split_min*:

```
join2 :: ('a × 'b) tree ⇒ ('a × 'b) tree ⇒ ('a × 'b) tree
join2 l r = (if r = ⟨⟩ then l else let (m, r') = split_min r in join l m r')
```

Function *split* splits a BST w.r.t. a given element a into a triple $(l,\ b,\ r)$ such that l contains the elements less than a, r contains the elements greater than a, and b is true iff a was in the input tree:

```
split :: 'a ⇒ ('a × 'b) tree ⇒ ('a × 'b) tree × bool × ('a × 'b) tree
split _ ⟨⟩ = (⟨⟩, False, ⟨⟩)
split x ⟨l, (a, _), r⟩
= (case cmp x a of
    LT ⇒ let (l₁, b, l₂) = split x l in (l₁, b, join l₂ a r) |
    EQ ⇒ (l, True, r) |
    GT ⇒ let (r₁, b, r₂) = split x r in (join l a r₁, b, r₂))
```

The following example demonstrates the workings of *split*:



Assume $a < b < c < d < e$. The call *split* c descends the input BST along the path a, e, b, d, splits the tree into two parts on each level and reassembles the parts into the two separate output trees on the way back up using *join*. For simplicity the example assumes that *join* just puts the subtrees together but no rebalancing is needed.

Insertion and deletion can be define in terms of *split* and *join*:

```
insert :: 'a ⇒ ('a × 'b) tree ⇒ ('a × 'b) tree
insert x t = (let (l, b, r) = split x t in join l x r)

delete :: 'a ⇒ ('a × 'b) tree ⇒ ('a × 'b) tree
delete x t = (let (l, b, r) = split x t in join2 l r)
```

Efficiency can be improved a little by taking the returned b into account (how?). Alternatively, insertion and deletion can be defined by means of union and difference (Exercise 10.1).

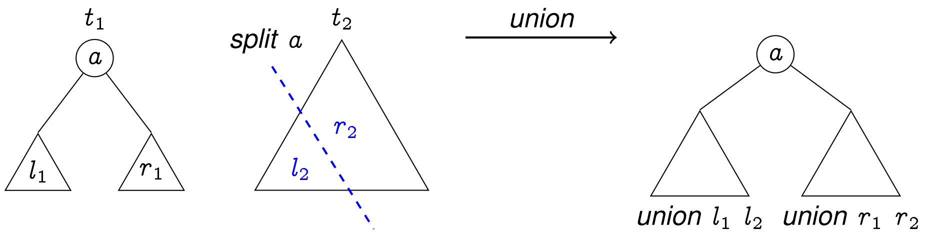
But we have bigger functions to fry: union, intersection and difference. They are shown in Figure 10.3. All three are divide-and-conquer algorithms that follow the same schema: both input trees are split at an element a (by construction or explicitly), the

```
union :: ('a × 'b) tree ⇒ ('a × 'b) tree ⇒ ('a × 'b) tree
union ⟨⟩ t = t
union t ⟨⟩ = t
union ⟨l₁, (a, _), r₁⟩ t₂
= (let (l₂, b, r₂) = split a t₂
   in join (union l₁ l₂) a (union r₁ r₂))

inter :: ('a × 'b) tree ⇒ ('a × 'b) tree ⇒ ('a × 'b) tree
inter ⟨⟩ t = ⟨⟩
inter t ⟨⟩ = ⟨⟩
inter ⟨l₁, (a, _), r₁⟩ t₂
= (let (l₂, b, r₂) = split a t₂;
       l' = inter l₁ l₂; r' = inter r₁ r₂
   in if b then join l' a r' else join2 l' r')

diff :: ('a × 'b) tree ⇒ ('a × 'b) tree ⇒ ('a × 'b) tree
diff ⟨⟩ t = ⟨⟩
diff t ⟨⟩ = t
diff t₁ ⟨l₂, (a, _), r₂⟩
= (let (l₁, b, r₁) = split a t₁
   in join2 (diff l₁ l₂) (diff r₁ r₂))
```

Figure 10.3 Union, intersection and difference

algorithm is applied recursively to the two trees of the elements below a and to the two trees of the elements above a, and the two results are suitably joined.

The following diagram demonstrates the behaviour of *union*:



10.2.1 Correctness

We need to prove that *union*, *inter* and *diff* satisfy the specification in Figure 10.1 where set = *set_tree* and $invar\ t = inv\ t \wedge \mathit{bst}\ t$. That is, for each function we show its set-theoretic property and preservation of inv and *bst* using the assumptions in Figure 10.2. Most of the proofs in this section are obvious and automatic inductions and we do not discuss them.

First we need to prove suitable properties of the auxiliary functions *split_min*, *join2* and *split*:

$$\mathit{split\_min}\ t = (m,\ t') \wedge t \neq \langle\rangle \longrightarrow$$
$$m \in \mathit{set\_tree}\ t \wedge \mathit{set\_tree}\ t = \{m\} \cup \mathit{set\_tree}\ t'$$

$$\mathit{split\_min}\ t = (m,\ t') \wedge \mathit{bst}\ t \wedge t \neq \langle\rangle \longrightarrow$$
$$\mathit{bst}\ t' \wedge (\forall x{\in}\mathit{set\_tree}\ t'.\ m < x)$$

$$\mathit{split\_min}\ t = (m,\ t') \wedge inv\ t \wedge t \neq \langle\rangle \longrightarrow inv\ t'$$

$$\mathit{set\_tree}\ (\mathit{join2}\ l\ r) = \mathit{set\_tree}\ l \cup \mathit{set\_tree}\ r \tag{10.5}$$

$$\mathit{bst}\ l \wedge \mathit{bst}\ r \wedge (\forall x{\in}\mathit{set\_tree}\ l.\ \forall y{\in}\mathit{set\_tree}\ r.\ x < y) \longrightarrow$$
$$\mathit{bst}\ (\mathit{join2}\ l\ r)$$

$$inv\ l \wedge inv\ r \longrightarrow inv\ (\mathit{join2}\ l\ r)$$

$$\mathit{split}\ x\ t = (l,\ b,\ r) \wedge \mathit{bst}\ t \longrightarrow$$
$$\mathit{set\_tree}\ l = \{a \in \mathit{set\_tree}\ t \mid a < x\} \wedge$$
$$\mathit{set\_tree}\ r = \{a \in \mathit{set\_tree}\ t \mid x < a\} \wedge$$
$$b = (x \in \mathit{set\_tree}\ t) \wedge \mathit{bst}\ l \wedge \mathit{bst}\ r \tag{10.6}$$

$$\mathit{split}\ x\ t = (l,\ b,\ r) \wedge inv\ t \longrightarrow inv\ l \wedge inv\ r$$

The correctness properties of *insert* and *delete* are trivial consequences and are not shown. We move on to *union*. Its correctness properties are concretizations of the properties (*union*) and (*union-inv*) in Figure 10.1:

$$\mathit{bst}\ t_2 \longrightarrow \mathit{set\_tree}\ (\mathit{union}\ t_1\ t_2) = \mathit{set\_tree}\ t_1 \cup \mathit{set\_tree}\ t_2$$

$$\mathit{bst}\ t_1 \wedge \mathit{bst}\ t_2 \longrightarrow \mathit{bst}\ (\mathit{union}\ t_1\ t_2)$$

$$inv\ t_1 \wedge inv\ t_2 \longrightarrow inv\ (\mathit{union}\ t_1\ t_2)$$

All three *union* properties are proved by computation induction. The first property follows easily from assumption (10.1) and (10.6). The assumption *bst* t_2 (but not *bst* t_1) is required because t_2 is split and (10.6) requires *bst*. Preservation of *bst* follows from assumption (10.2) with the help of the first *union* property and the preservation of *bst* by *split*. Preservation of inv follows from assumptions (10.3) and (10.4) with the help of the preservation of inv by *split*.

The correctness properties of *inter* look similar:

$$\mathit{bst}\ t_1 \wedge \mathit{bst}\ t_2 \longrightarrow \mathit{set\_tree}\ (\mathit{inter}\ t_1\ t_2) = \mathit{set\_tree}\ t_1 \cap \mathit{set\_tree}\ t_2$$
$$\mathit{bst}\ t_1 \wedge \mathit{bst}\ t_2 \longrightarrow \mathit{bst}\ (\mathit{inter}\ t_1\ t_2)$$
$$\mathit{inv}\ t_1 \wedge \mathit{inv}\ t_2 \longrightarrow \mathit{inv}\ (\mathit{inter}\ t_1\ t_2)$$

The proof of the preservation properties are automatic but the proof of the *set_tree* property is more involved than the corresponding proof for *union* and we take a closer look at the induction. We focus on the case $t_1 = \langle l_1, (a, \_), r_1\rangle$ and $t_2 \neq \langle\rangle$. Let $L_1 = \mathit{set\_tree}\ l_1$ and $R_1 = \mathit{set\_tree}\ r_1$. Let $(l_2, b, r_2) = \mathit{split}\ t_2\ a$, $L_2 = \mathit{set\_tree}\ l_2$, $R_2 = \mathit{set\_tree}\ r_2$ and $A = (\textbf{if}\ b\ \textbf{then}\ \{a\}\ \textbf{else}\ \{\})$. The separation properties

$$a \notin L_1 \cup R_1 \qquad a \notin L_2 \cup R_2$$
$$L_2 \cap R_2 = \{\} \qquad L_1 \cap R_2 = \{\} \qquad L_2 \cap R_1 = \{\}$$

follow from *bst* t_1, *bst* t_2 and (10.6). Now for the main proof:

$$\begin{array}{ll} \mathit{set\_tree}\ t_1 \cap \mathit{set\_tree}\ t_2 & \\ = (L_1 \cup R_1 \cup \{a\}) \cap (L_2 \cup R_2 \cup A) & \text{by (10.6), } \mathit{bst}\ t_2 \\ = L_1 \cap L_2 \cup R_1 \cap R_2 \cup A & \text{by the separation properties} \\ = \mathit{set\_tree}\ (\mathit{inter}\ t_1\ t_2) & \text{by (10.1), (10.5), IHs, } \mathit{bst}\ t_1,\ \mathit{bst}\ t_2, \text{(10.6)} \end{array}$$

The correctness properties of *diff* follow the same pattern and their proofs are similar to the proofs of the *inter* properties. This concludes the generic join approach.

10.3 Joining Red-Black Trees

This section shows how to implement *join* efficiently on red-black trees. The basic idea is simple: descend along the spine of the higher of the two trees until reaching a subtree whose height is the same as the height of the lower tree. With suitable changes this works for other balanced trees as well [Blelloch et al. 2022]. The function definitions are shown in Figure 10.4. Function *join* calls *joinR* (descending along the right spine of l) if l is the higher tree, or calls *joinL* (descending along the left spine of r) if r is the higher tree, or returns $B\ l\ x\ r$ otherwise. The running time is linear in the black height (and thus logarithmic in the size) if we assume that the black height is stored in each node; our implementation of red-black trees would have to be augmented accordingly. Note that in *joinR* (and similarly in *joinL*) the comparison is not *bh* l = *bh* r but *bh* $l \leq$ *bh* r to simplify the proofs.

10.3.1 Correctness

We need to prove that *join* on red-black trees (and a suitable *inv*) satisfies its specification in Figure 10.2. We start with properties of *joinL*; the properties of function *joinR* are completely symmetric. These are the three automatically provable inductive propositions:

```
joinL :: 'a rbt ⇒ 'a ⇒ 'a rbt ⇒ 'a rbt
joinL l x r
= (if bh r ≤ bh l then R l x r
   else case r of
        ⟨l', (x', Red), r'⟩ ⇒ R (joinL l x l') x' r' |
        ⟨l', (x', Black), r'⟩ ⇒ baliL (joinL l x l') x' r')

joinR :: 'a rbt ⇒ 'a ⇒ 'a rbt ⇒ 'a rbt
joinR l x r
= (if bh l ≤ bh r then R l x r
   else case l of
        ⟨l', (x', Red), r'⟩ ⇒ R l' x' (joinR r' x r) |
        ⟨l', (x', Black), r'⟩ ⇒ baliR l' x' (joinR r' x r))

join :: 'a rbt ⇒ 'a ⇒ 'a rbt ⇒ 'a rbt
join l x r
= (if bh r < bh l then paint Black (joinR l x r)
   else if bh l < bh r then paint Black (joinL l x r) else B l x r)
```

Figure 10.4 Function *join* on red-black trees

$$\begin{aligned}&\mathit{invc}\ l \wedge \mathit{invc}\ r \wedge \mathit{invh}\ l \wedge \mathit{invh}\ r \wedge \mathit{bh}\ l \le \mathit{bh}\ r \longrightarrow\\ &\mathit{invc2}\ (\mathit{joinL}\ l\ x\ r) \wedge\\ &(\mathit{bh}\ l \neq \mathit{bh}\ r \wedge \mathit{color}\ r = \mathit{Black} \longrightarrow \mathit{invc}\ (\mathit{joinL}\ l\ x\ r)) \wedge\\ &\mathit{invh}\ (\mathit{joinL}\ l\ x\ r) \wedge \mathit{bh}\ (\mathit{joinL}\ l\ x\ r) = \mathit{bh}\ r\end{aligned}$$

$$\mathit{bh}\ l \le \mathit{bh}\ r \longrightarrow \mathit{set\_tree}\ (\mathit{joinL}\ l\ x\ r) = \mathit{set\_tree}\ l \cup \{x\} \cup \mathit{set\_tree}\ r$$

$$\mathit{bst}\ \langle l,\ (a,\ n),\ r\rangle \wedge \mathit{bh}\ l \le \mathit{bh}\ r \longrightarrow \mathit{bst}\ (\mathit{joinL}\ l\ a\ r)$$

Because *joinL* employs *baliL* from the chapter on red-black trees, the proof of the first proposition makes use of the property of *baliL* displayed in Section 8.2.1.

We define the invariant *inv* required by the specification in Figure 10.2 as follows:

$$\mathit{inv}\ t = (\mathit{invc}\ t \wedge \mathit{invh}\ t)$$

Although weaker than *rbt*, it still guarantees logarithmic height (Exercise 8.1). Note that *rbt* itself does not work because it does not satisfy property (10.4). The properties of *join* and *inv* are now easy consequences of the *joinL* (and *joinR*) properties shown above.

10.4 Exercises

Exercise 10.1. Define alternative versions *insert'* and *delete'* of *insert* and *delete* using *union* and *diff* (and *join* and $\langle\rangle$). Prove their correctness as in Section 10.2.1: *set_tree* yields the right result and *bst* is preserved.

Exercise 10.2. Define an alternative version *diff1* of *diff* where in the third equation pattern matching is on t_1 and t_2 is *split*. Prove that *bst* $t_1 \wedge$ *bst* t_2 implies both *set_tree* (*diff1* t_1 t_2) = *set_tree* t_1 − *set_tree* t_2 and *bst* (*diff1* t_1 t_2).

Exercise 10.3. Following the general idea of the join function for red-black trees, define a join function for 2-3-trees. Start with two functions *joinL*, *joinR* :: $'a\ tree23 \Rightarrow\ 'a \Rightarrow\ 'a\ tree23 \Rightarrow\ 'a\ up_i$ and combine them into the overall join function:

join :: $'a\ tree23 \Rightarrow\ 'a \Rightarrow\ 'a\ tree23 \Rightarrow\ 'a\ tree23$

Prove the following correctness properties:

complete $l \wedge$ *complete* $r \longrightarrow$ *complete* (*join* $l\ x\ r$)

complete $l \wedge$ *complete* $r \longrightarrow$
inorder (*join* $l\ x\ r$) = *inorder* l @ x # *inorder* r

The corresponding (and needed) properties of *joinL* and *joinR* are slightly more involved.

Chapter Notes

The join approach goes back to Adams [1993]. Blelloch et al. [2022] generalized the approach from weight-balanced trees to AVL trees, red-black trees and treaps. In particular they proved the $O(m \lg(\frac{n}{m}+1))$ complexity bound.

11 Arrays via Braun Trees

Tobias Nipkow

Braun trees are a subclass of almost complete trees. In this chapter we explore their use as arrays and in Chapter 16 as priority queues.

11.1 Array

So far we have discussed sets (or maps) over some arbitrary linearly ordered type. Now we specialize that linearly ordered type to *nat* to model arrays. In principle we could model arrays as maps from a subset of natural numbers to the array elements. Because arrays are contiguous, it is more appropriate to model them as lists. The type *'a list* comes with two array-like operations (see Appendix A):

Indexing: $xs\ !\ n$ is the nth element of the list xs.

Updating: $xs[n := x]$ is xs with the nth element replaced by x.

By convention, indexing starts with $n = 0$. If $n \geq |xs|$ then $xs\ !\ n$ and $xs[n := x]$ are underdefined: they are defined terms but we do not know what their value is.

Note that operationally, indexing and updating take time linear in the index, which may appear inappropriate for arrays. However, the type of lists is only an abstract model that specifies the desired functional behaviour of arrays, but not their running time complexity.

The ADT of arrays is shown in Figure 11.1. Type *'ar* is the type of arrays, type *'a* the type of elements in the arrays. The abstraction function *list* abstracts arrays to lists. It would make perfect sense to include *list* in the interface as well. In fact, our implementation below comes with a (reasonably efficiently) executable definition of *list*.

The behaviour of *lookup*, *update*, *size* and *array* is specified in terms of their counterparts on lists and requires that the invariant is preserved. What distinguishes the specifications of *lookup* and *update* from the standard schema (see Chapter 6) is that they carry a size precondition because the result of *lookup* and *update* is only specified if the index is less than the size of the array.

ADT *Array* =

interface

$lookup :: 'ar \Rightarrow nat \Rightarrow 'a$

$update :: nat \Rightarrow 'a \Rightarrow 'ar \Rightarrow 'ar$

$len :: 'ar \Rightarrow nat$

$array :: 'a\ list \Rightarrow 'ar$

abstraction $list :: 'ar \Rightarrow 'a\ list$

invariant $invar :: 'ar \Rightarrow bool$

specification

$invar\ ar \wedge n < len\ ar \longrightarrow lookup\ ar\ n = list\ ar\ !\ n$ (*lookup*)

$invar\ ar \wedge n < len\ ar \longrightarrow list\ (update\ n\ x\ ar) = (list\ ar)[n := x]$ (*update*)

$invar\ ar \wedge n < len\ ar \longrightarrow invar\ (update\ n\ x\ ar)$ (*update-inv*)

$invar\ ar \longrightarrow len\ ar = |list\ ar|$ (*len*)

$list\ (array\ xs) = xs$ (*array*)

$invar\ (array\ xs)$ (*array-inv*)

Figure 11.1 ADT *Array*

11.2 Braun Trees

One can implement arrays by any one of the many search trees presented in this book. Instead we take advantage of the fact that the keys are natural numbers and implement arrays by so-called **Braun trees** that are almost complete and thus have minimal height.

The basic idea is to index a node in a binary tree by the non-zero bit string that leads from the root to that node in the following fashion. Starting from the least significant bit and while we have not reached the leading 1 (which is ignored), we examine the bits one by one. If the current bit is 0, descend into the left child, otherwise into the right child. Instead of bit strings we use the natural numbers ≥ 1 that they represent. The Braun tree with nodes indexed by 1–15 is shown in Figure 11.2. The numbers are the indexes and not the elements stored in the nodes. For example, the index 14 is 0111 in binary (least significant bit first). If you follow the path left-right-right (corresponding to 011) in Figure 11.2, you reach node 14.

A tree t is suitable for representing an array if the set of indexes of all its nodes is the interval $\{1..|t|\}$. The following tree is unsuitable because the node indexed by 2 is missing:

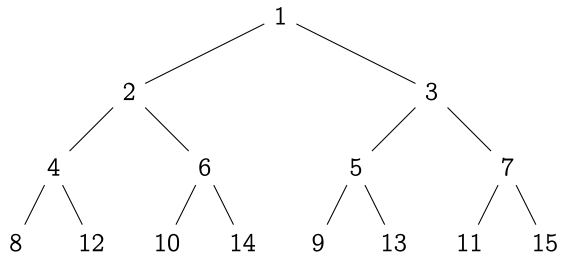
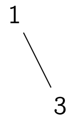


Figure 11.2 Braun tree with nodes indexed by 1–15



It turns out that the following invariant guarantees that a tree t contains exactly the nodes indexed by 1, ..., $|t|$:

$$\mathit{braun} :: \ 'a\ \mathit{tree} \Rightarrow \mathit{bool}$$

$$\mathit{braun}\ \langle\rangle = \mathit{True}$$
$$\mathit{braun}\ \langle l,\ \_,\ r\rangle = ((|l| = |r| \lor |l| = |r| + 1) \land \mathit{braun}\ l \land \mathit{braun}\ r)$$

The disjunction can alternatively be expressed as $|r| \le |l| \le |r| + 1$. We call a tree a **Braun tree** iff it satisfies predicate *braun*.

Although we do not need or prove this here, it is interesting to note that a tree that contains exactly the nodes indexed by 1, ..., $|t|$ is a Braun tree.

Let us now prove the earlier claim that Braun trees are almost complete. First, a lemma about the composition of almost complete trees:

Lemma 11.1. $\mathit{acomplete}\ l \land \mathit{acomplete}\ r \land |l| = |r| + 1 \longrightarrow \mathit{acomplete}\ \langle l,\ x,\ r\rangle$

Proof. Using Lemmas 4.7 and 4.8 and the assumptions we obtain

$$h\ \langle l,\ x,\ r\rangle = \lceil \mathit{lg}\ (|r|_1 + 1)\rceil + 1 \qquad (*)$$
$$\mathit{mh}\ \langle l,\ x,\ r\rangle = \lfloor \mathit{lg}\ |r|_1\rfloor + 1 \qquad (**)$$

Because $1 \le |r|_1$ there is an i such that $2^i \le |r|_1 < 2^{i\,+\,1}$ and thus $2^i < |r|_1 + 1 \le 2^{i\,+\,1}$. This implies $i = \lfloor \mathit{lg}\ |r|_1\rfloor$ and $i + 1 = \lceil \mathit{lg}\ (|r|_1 + 1)\rceil$. Together with $(*)$ and $(**)$ this implies $\mathit{acomplete}\ \langle l,\ x,\ r\rangle$. □

Now we can show that all Braun trees are almost complete. Thus we know that they have optimal height (Lemma 4.6) and can even quantify it (Lemma 4.7).

Lemma 11.2. *braun* $t \longrightarrow$ *acomplete* t

Proof by induction. We focus on the induction step where $t = \langle l, x, r\rangle$. Because of *braun* t we can distinguish two cases. First assume $|l| = |r| + 1$. The claim *acomplete* t follows immediately from the previous lemma. Now assume $|l| = |r|$. By definition, there are four cases to consider when proving *acomplete* t. By symmetry it suffices to consider only two of them. If *h* $l \le$ *h* r and *mh* $r <$ *mh* l then *acomplete* t reduces to *acomplete* r, which is true by IH. Now assume *h* $l \le$ *h* r and *mh* $l \le$ *mh* r. Because $|l| = |r|$, the fact that the height of an almost complete tree is determined uniquely by its size (Lemma 4.7) implies *h* $l =$ *h* r and thus *acomplete* t reduces to *acomplete* l, which is again true by IH. □

Note that the proof does not rely on the fact that it is the left child that is potentially one bigger than the right one; it merely requires that the difference in size between two siblings is at most 1.

11.3 Arrays via Braun Trees

In this section we implement arrays via Braun trees and verify correctness and complexity. We start by defining array-like functions on Braun trees. After the above explanation of Braun trees the following lookup function will not come as a surprise:

```
lookup1 :: 'a tree ⇒ nat ⇒ 'a
lookup1 ⟨l, x, r⟩ n
= (if n = 1 then x else lookup1 (if even n then l else r) (n div 2))
```

The least significant bit is the parity of the index and we advance to the next bit by div 2. The function is called *lookup1* rather than *lookup* to emphasize that it expects the index to be at least 1. This simplifies the implementation via Braun trees but is in contrast to the *Array* interface where by convention indexing starts with 0.

Function *update1* descends in the very same manner but also performs an update when reaching 1:

```
update1 :: nat ⇒ 'a ⇒ 'a tree ⇒ 'a tree
update1 _ x ⟨⟩ = ⟨⟨⟩, x, ⟨⟩⟩
update1 n x ⟨l, a, r⟩
= (if n = 1 then ⟨l, x, r⟩
   else if even n then ⟨update1 (n div 2) x l, a, r⟩
        else ⟨l, a, update1 (n div 2) x r⟩)
```

$$\begin{array}{lcl}
\mathit{lookup}\ (t,\ \_)\ n & = & \mathsf{lookup1}\ t\ (n + 1) \\
\mathit{update}\ n\ x\ (t,\ m) & = & (\mathsf{update1}\ (n + 1)\ x\ t,\ m) \\
\mathit{len}\ (t,\ m) & = & m \\
\mathit{array}\ xs & = & (\mathsf{adds}\ xs\ 0\ \langle\rangle,\ |xs|)
\end{array}$$

Figure 11.3 Array implementation via Braun trees

The second equation updates existing entries in case $n = 1$. The first equation, however, creates a new entry and thus supports extending the tree. That is, *update1* $(|t| + 1)\ x\ t$ extends the tree with a new node x at index $|t| + 1$. Function *adds* iterates this process (again expecting $|t| + 1$ as the index) and thus adds a whole list of elements:

$$\mathsf{adds} :: {}'a\ \mathit{list} \Rightarrow \mathit{nat} \Rightarrow {}'a\ \mathit{tree} \Rightarrow {}'a\ \mathit{tree}$$

$$\mathsf{adds}\ []\ \_\ t = t$$
$$\mathsf{adds}\ (x\ \#\ xs)\ n\ t = \mathsf{adds}\ xs\ (n + 1)\ (\mathsf{update1}\ (n + 1)\ x\ t)$$

The implementation of the *Array* interface in Figure 11.3 is just a thin wrapper around the corresponding functions on Braun trees. An array is represented as a pair of a Braun tree and its size. Note that although *update1* can extend the tree, the specification and implementation of the array *update* function does not support that: n is expected to be below the length of the array. Flexible arrays are specified and implemented in Section 11.4.

11.3.1 Correctness

The invariant on arrays is obvious:

$$\mathit{invar}\ (t,\ l) = (\mathsf{braun}\ t \wedge l = |t|)$$

The abstraction function *list* delegates to a namesake *list* on trees:

$$\mathit{list}\ (t,l) = \mathsf{list}\ t$$

Function *list* could be defined in the following intuitive way, where $[m..{<}n]$ is the list of natural numbers from m up to but excluding n (see Appendix A):

$$\textit{list}\ t = \textit{map}\ (\textit{lookup1}\ t)\ [1..<|t| + 1]$$

Instead we define *list* recursively and derive the above equation later on

$$\begin{aligned}
&\textit{list} :: \text{'}a\ \textit{tree} \Rightarrow \text{'}a\ \textit{list}\\
&\textit{list}\ \langle\rangle = []\\
&\textit{list}\ \langle l,\ x,\ r\rangle = x\ \#\ \textit{splice}\ (\textit{list}\ l)\ (\textit{list}\ r)
\end{aligned}$$

This definition is best explained by looking at Figure 11.2. The subtrees with root 2 and 3 will be mapped to the lists [2, 4, 6, 8, 10, 12, 14] and [1, 3, 5, 7, 9, 11, 13, 15]. The obvious way to combine these two lists into [1, 2, 3, ..., 15] is to splice them:

$$\begin{aligned}
&\textit{splice} :: \text{'}a\ \textit{list} \Rightarrow \text{'}a\ \textit{list} \Rightarrow \text{'}a\ \textit{list}\\
&\textit{splice}\ []\ ys = ys\\
&\textit{splice}\ (x\ \#\ xs)\ ys = x\ \#\ \textit{splice}\ ys\ xs
\end{aligned}$$

Note that because of this reasonably efficient ($O(n \lg n)$, see Section 11.3.2) implementation of *list* we can also regard *list* as part of the interface of arrays.

Before we embark on the actual proofs we state a helpful arithmetic truth that is frequently used implicitly below:

$$\begin{aligned}
&\textit{braun}\ \langle l,\ x,\ r\rangle \wedge n \in \{1..|\langle l,\ x,\ r\rangle|\} \wedge 1 < n \longrightarrow\\
&(\textit{odd}\ n \longrightarrow n\ \text{div}\ 2 \in \{1..|r|\}) \wedge (\textit{even}\ n \longrightarrow n\ \text{div}\ 2 \in \{1..|l|\})
\end{aligned}$$

where $\{m..n\} = \{k \mid m \le k \wedge k \le m\}$.

We will now verify that the implementation in Figure 11.3 of the *Array* interface in Figure 11.1 satisfies the given specification.

We start with proposition (*len*), the correctness of function *len*. Because of the invariant, (*len*) follows directly from

$$|\textit{list}\ t| = |t|$$

which is proved by induction. This fact is used implicitly in many proofs below.

The following proposition implies the correctness property (*lookup*):

$$\textit{braun}\ t \wedge i < |t| \longrightarrow \textit{list}\ t\ !\ i = \textit{lookup1}\ t\ (i + 1) \tag{11.1}$$

The proof is by induction and uses the following proposition that is also proved by induction:

$$\begin{aligned}
&n < |xs| + |ys| \wedge |ys| \le |xs| \wedge |xs| \le |ys| + 1 \longrightarrow\\
&\textit{splice}\ xs\ ys\ !\ n = (\textbf{if}\ \textit{even}\ n\ \textbf{then}\ xs\ \textbf{else}\ ys)\ !\ (n\ \text{div}\ 2)
\end{aligned}$$

As a corollary to (11.1) we obtain that function *list* can indeed be expressed via *lookup1*:

$$\mathit{braun}\ t \longrightarrow \mathit{list}\ t = \mathit{map}\ (\mathit{lookup1}\ t)\ [1..<|t| + 1] \tag{11.2}$$

It follows by **list extensionality**:

$$xs = ys \longleftrightarrow |xs| = |ys| \wedge (\forall i{<}|xs|.\ xs\ !\ i = ys\ !\ i)$$

Let us now verify *update* as implemented via *update1*. The following two preservation properties (proved by induction) prove (*update-inv*):

$$\mathit{braun}\ t \wedge n \in \{1..|t|\} \longrightarrow |\mathit{update1}\ n\ x\ t| = |t|$$

$$\mathit{braun}\ t \wedge n \in \{1..|t|\} \longrightarrow \mathit{braun}\ (\mathit{update1}\ n\ x\ t)$$

The following property relating *lookup1* and *update1* is again proved by induction:

$$\begin{array}{l}\mathit{braun}\ t \wedge n \in \{1..|t|\} \longrightarrow \\ \mathit{lookup1}\ (\mathit{update1}\ n\ x\ t)\ m = (\textbf{if}\ n = m\ \textbf{then}\ x\ \textbf{else}\ \mathit{lookup1}\ t\ m)\end{array}$$

The last three properties together with (11.2) and list extensionality prove the following proposition, which implies (*update*):

$$\mathit{braun}\ t \wedge n \in \{1..|t|\} \longrightarrow \mathit{list}\ (\mathit{update1}\ n\ x\ t) = (\mathit{list}\ t)[n - 1 := x]$$

Finally we turn to the constructor *array*. It is implemented in terms of *adds* and *update1*. Their correctness is captured by the following properties whose inductive proofs build on each other:

$$\mathit{braun}\ t \longrightarrow |\mathit{update1}\ (|t| + 1)\ x\ t| = |t| + 1 \tag{11.3}$$

$$\mathit{braun}\ t \longrightarrow \mathit{braun}\ (\mathit{update1}\ (|t| + 1)\ x\ t) \tag{11.4}$$

$$\mathit{braun}\ t \longrightarrow \mathit{list}\ (\mathit{update1}\ (|t| + 1)\ x\ t) = \mathit{list}\ t\ @\ [x] \tag{11.5}$$

$$\mathit{braun}\ t \longrightarrow |\mathit{adds}\ xs\ |t|\ t| = |t| + |xs| \wedge \mathit{braun}\ (\mathit{adds}\ xs\ |t|\ t)$$

$$\mathit{braun}\ t \longrightarrow \mathit{list}\ (\mathit{adds}\ xs\ |t|\ t) = \mathit{list}\ t\ @\ xs$$

The last two properties imply the remaining proof obligations (*array*) and (*array-inv*). The proof of (11.5) requires the following two properties of *splice* which are proved by simultaneous induction:

$$\begin{array}{l}|ys| \le |xs| \longrightarrow \mathit{splice}\ (xs\ @\ [x])\ ys = \mathit{splice}\ xs\ ys\ @\ [x] \\ |xs| \le |ys| + 1 \longrightarrow \mathit{splice}\ xs\ (ys\ @\ [y]) = \mathit{splice}\ xs\ ys\ @\ [y]\end{array}$$

11.3.2 Running Time

The running time of *lookup1* and *update1* is obviously logarithmic because of the logarithmic height of Braun trees. We sketch why *list* and *array* both have running time $O(n \lg n)$. Linear time versions are presented in Section 11.5.

Function *list* is similar to bottom-up merge sort and *splice* is similar to *merge*. We focus on *splice* because it performs almost all the work. Consider calling *list* on a complete tree of height h. At each level k (starting with 0 for the root) of the tree, *splice* is called 2^k times with lists of size (almost) 2^{h-k-1}. The running time of *splice* with lists of the same length is proportional to the size of the lists. Thus the running time at each level is $O(2^k 2^{h-k-1}) = O(2^{h-1}) = O(2^h)$. Thus all the splices together require time $O(h 2^h)$. Because complete trees have size $n = 2^h$, the bound $O(n \lg n)$ follows.

Function *array* is implemented via *adds* and thus via repeated calls of *update1*. At the beginning of Section 7.3 we show that because *update1* has logarithmic complexity, calling it n times on a growing tree starting with a leaf takes time $\Theta(n \lg n)$.

11.4 Flexible Arrays

Flexible arrays can be grown and shrunk at either end. Figure 11.4 shows the specification of all four operations. (For *tl* and *butlast* see Appendix A.) *Array_Flex* extends the basic *Array* in Figure 11.1.

Below we first implement the *Array_Flex* functions on Braun trees. In a final step an implementation of *Array_Flex* on (tree, size) pairs is derived.

We have already seen that *update1* adds an element at the high end. The inverse operation *del_hi* removes the high end, assuming that the given index is the size of the tree:

```
del_hi :: nat ⇒ 'a tree ⇒ 'a tree
del_hi _ ⟨⟩ = ⟨⟩
del_hi n ⟨l, x, r⟩
= (if n = 1 then ⟨⟩
   else if even n then ⟨del_hi (n div 2) l, x, r⟩
        else ⟨l, x, del_hi (n div 2) r⟩)
```

This was easy but extending an array at the low end seems hard because one has to shift the existing entries. However, Braun trees support a logarithmic implementation:

ADT $\mathit{Array\_Flex} = \mathit{Array}\ +$

interface

$\mathit{add\_lo} :: {}'a \Rightarrow {}'ar \Rightarrow {}'ar$
$\mathit{del\_lo} :: {}'ar \Rightarrow {}'ar$
$\mathit{add\_hi} :: {}'a \Rightarrow {}'ar \Rightarrow {}'ar$
$\mathit{del\_hi} :: {}'ar \Rightarrow {}'ar$

specification

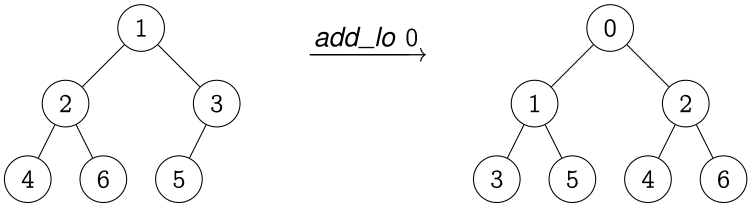
$\mathit{invar}\ ar \longrightarrow \mathit{invar}\ (\mathit{add\_lo}\ a\ ar)$ $(\mathit{add\_lo\text{-}inv})$
$\mathit{invar}\ ar \longrightarrow \mathit{list}\ (\mathit{add\_lo}\ a\ ar) = a\ \#\ \mathit{list}\ ar$ $(\mathit{add\_lo})$
$\mathit{invar}\ ar \longrightarrow \mathit{invar}\ (\mathit{del\_lo}\ ar)$ $(\mathit{del\_lo\text{-}inv})$
$\mathit{invar}\ ar \longrightarrow \mathit{list}\ (\mathit{del\_lo}\ ar) = \mathsf{tl}\ (\mathit{list}\ ar)$ $(\mathit{del\_lo})$
$\mathit{invar}\ ar \longrightarrow \mathit{invar}\ (\mathit{add\_hi}\ a\ ar)$ $(\mathit{add\_hi\text{-}inv})$
$\mathit{invar}\ ar \longrightarrow \mathit{list}\ (\mathit{add\_hi}\ a\ ar) = \mathit{list}\ ar\ @\ [a]$ $(\mathit{add\_hi})$
$\mathit{invar}\ ar \longrightarrow \mathit{invar}\ (\mathit{del\_hi}\ ar)$ $(\mathit{del\_hi\text{-}inv})$
$\mathit{invar}\ ar \longrightarrow \mathit{list}\ (\mathit{del\_hi}\ ar) = \mathsf{butlast}\ (\mathit{list}\ ar)$ $(\mathit{del\_hi})$

Figure 11.4 ADT *Array_Flex*

$\mathsf{add\_lo} :: {}'a \Rightarrow {}'a\ \mathit{tree} \Rightarrow {}'a\ \mathit{tree}$

$\mathsf{add\_lo}\ x\ \langle\rangle = \langle\langle\rangle,\ x,\ \langle\rangle\rangle$
$\mathsf{add\_lo}\ x\ \langle l,\ a,\ r\rangle = \langle \mathsf{add\_lo}\ a\ r,\ x,\ l\rangle$

The intended functionality is $\mathsf{list}\ (\mathsf{add\_lo}\ x\ t) = x\ \#\ \mathsf{list}\ t$. Function *add_lo* installs the new element x at the root of the tree. Because *add_lo* needs to shift the indices of the elements already in the tree, the left child (indices 2, 4, . . .) becomes the new right child (indices 3, 5, . . .). The old right child becomes the new left child with the old root a added in at index 2 and the remaining elements at indices 4, 6, ... In the following example, *add_lo* 0 transforms the left tree into the right one. The numbers in the nodes are the actual elements, not their indices.



$$\begin{array}{lcl}
add\_lo\ x\ (t,\ l) & = & (\textsf{add\_lo}\ x\ t,\ l + 1) \\
del\_lo\ (t,\ l) & = & (\textsf{del\_lo}\ t,\ l - 1) \\
add\_hi\ x\ (t,\ l) & = & (\textsf{update1}\ (l + 1)\ x\ t,\ l + 1) \\
del\_hi\ (t,\ l) & = & (\textsf{del\_hi}\ l\ t,\ l - 1)
\end{array}$$

Figure 11.5 Flexible array implementation via Braun trees

Function *del_lo* simply reverses *add_lo* by removing the root and merging the children:

$$\begin{array}{l}
\textsf{del\_lo} :: {'a}\ tree \Rightarrow {'a}\ tree \\
\textsf{del\_lo}\ \langle\rangle = \langle\rangle \\
\textsf{del\_lo}\ \langle l,\ \_,\ r\rangle = \textsf{merge}\ l\ r \\
\\
\textsf{merge} :: {'a}\ tree \Rightarrow {'a}\ tree \Rightarrow {'a}\ tree \\
\textsf{merge}\ \langle\rangle\ r = r \\
\textsf{merge}\ \langle l,\ a,\ r\rangle\ rr = \langle rr,\ a,\ \textsf{merge}\ l\ r\rangle
\end{array}$$

Figure 11.5 shows the obvious implementation of the functions in the *Array_Flex* interface in Figure 11.4 (on the left-hand side) with the help of the corresponding Braun tree operations (on the right-hand side). It is an extension of the basic array implementation from Figure 11.3. All *Array_Flex* functions have logarithmic time complexity because the corresponding Braun tree functions do because they descend along one branch of the tree.

11.4.1 Correctness

We now have to prove the properties in Figure 11.4. We have already dealt with *update1* and thus *add_hi* above. Properties (*add_hi-inv*) and (*add_hi*) follow from (11.3), (11.4) and (11.5) stated earlier.

Correctness of *del_hi* on Braun trees is captured by the following two properties proved by induction:

$$\textsf{braun}\ t \longrightarrow \textsf{braun}\ (\textsf{del\_hi}\ |t|\ t)$$

$$\textsf{braun}\ t \longrightarrow \textsf{list}\ (\textsf{del\_hi}\ |t|\ t) = \textsf{butlast}\ (\textsf{list}\ t) \tag{11.6}$$

They imply (*del_hi*) and (*del_hi-inv*). The proof of (11.6) requires the following property of *splice*, which is proved by induction:

$$\mathit{butlast}\ (\mathit{splice}\ xs\ ys)$$
$$= (\textbf{if}\ |ys| < |xs|\ \textbf{then}\ \mathit{splice}\ (\mathit{butlast}\ xs)\ ys\ \textbf{else}\ \mathit{splice}\ xs\ (\mathit{butlast}\ ys))$$

Correctness of *add_lo* on Braun trees (properties (*add_ lo*) and (*add_ lo-inv*)) follows directly from the following two inductive properties:

$$\mathit{braun}\ t \longrightarrow \mathit{list}\ (\mathit{add\_lo}\ a\ t) = a\ \#\ \mathit{list}\ t$$
$$\mathit{braun}\ t \longrightarrow \mathit{braun}\ (\mathit{add\_lo}\ x\ t)$$

Finally we turn to *del_lo*. Inductions (for *merge*) and case analyses (for *del_lo*) yield the following properties:

$$\mathit{braun}\ \langle l,\ x,\ r\rangle \longrightarrow \mathit{list}\ (\mathit{merge}\ l\ r) = \mathit{splice}\ (\mathit{list}\ l)\ (\mathit{list}\ r)$$
$$\mathit{braun}\ \langle l,\ x,\ r\rangle \longrightarrow \mathit{braun}\ (\mathit{merge}\ l\ r)$$
$$\mathit{braun}\ t \longrightarrow \mathit{list}\ (\mathit{del\_lo}\ t) = \mathit{tl}\ (\mathit{list}\ t)$$
$$\mathit{braun}\ t \longrightarrow \mathit{braun}\ (\mathit{del\_lo}\ t)$$

The last two properties imply (*del_ lo*) and (*del_ lo-inv*).

11.5 Bigger, Better, Faster, More!

In this section we meet efficient versions of some old and new functions on Braun trees. The implementation of the corresponding array operations is trivial and is not discussed.

11.5.1 Fast Size of Braun Trees

The size of a Braun tree can be computed without having to traverse the entire tree:

```
size_fast :: 'a tree ⇒ nat
size_fast ⟨⟩ = 0
size_fast ⟨l, _, r⟩ = (let n = size_fast r in 1 + 2 · n + diff l n)

diff :: 'a tree ⇒ nat ⇒ nat
diff ⟨⟩ _ = 0
diff ⟨l, _, r⟩ n
= (if n = 0 then 1
    else if even n then diff r (n div 2 − 1) else diff l (n div 2))
```

Function *size_fast* descends down the right spine, computes the size of a *Node* as if both children were the same size ($1 + 2 \cdot n$), but adds $\mathit{diff}\ l\ n$ to compensate for bigger left children. Correctness of *size_fast*

Lemma 11.3. $\mathit{braun}\ t \longrightarrow \mathit{size\_fast}\ t = |t|$

follows from this property of *diff*:

$$\mathit{braun}\ t \wedge |t| \in \{n,\ n + 1\} \longrightarrow \mathit{diff}\ t\ n = |t| - n$$

The running time of *size_fast* is quadratic in the height of the tree (Exercise 11.3).

11.5.2 Initializing a Braun Tree with a Fixed Value

Above we only considered the construction of a Braun tree from a list. Alternatively one may want to create a tree (array) where all elements are initialized to the same value. Of course one can call *update1* n times, but one can also build the tree directly:

```
braun_of_naive x n
= (if n = 0 then ⟨⟩
   else let m = (n − 1) div 2
        in if odd n
           then ⟨braun_of_naive x m, x, braun_of_naive x m⟩
           else ⟨braun_of_naive x (m + 1), x,
                 braun_of_naive x m⟩)
```

This solution also has time complexity $O(n \lg n)$ but it can clearly be improved by sharing identical recursive calls. Function *braun2_of* shares as much as possible by producing trees of size n and $n+1$ in parallel:

```
braun2_of :: 'a ⇒ nat ⇒ 'a tree × 'a tree
braun2_of x n
= (if n = 0 then (⟨⟩, ⟨⟨⟩, x, ⟨⟩⟩)
   else let (s, t) = braun2_of x ((n − 1) div 2)
        in if odd n then (⟨s, x, s⟩, ⟨t, x, s⟩) else (⟨t, x, s⟩, ⟨t, x, t⟩))

braun_of :: 'a ⇒ nat ⇒ 'a tree
braun_of x n = fst (braun2_of x n)
```

The running time is clearly logarithmic.

The correctness properties (see Appendix A for *replicate*)

$$\mathit{list}\ (\mathit{braun\_of}\ x\ n) = \mathit{replicate}\ n\ x$$

$$\mathit{braun}\ (\mathit{braun\_of}\ x\ n)$$

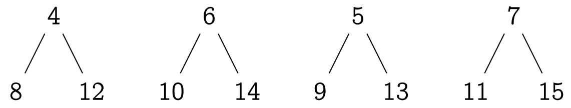
are corollaries of the more general statements which can be proved by induction:

$$\textit{braun2\_of}\ x\ n = (s,\ t) \longrightarrow$$
$$\textit{list}\ s = \textit{replicate}\ n\ x \wedge \textit{list}\ t = \textit{replicate}\ (n + 1)\ x$$

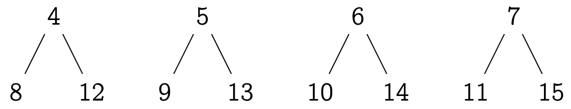
$$\textit{braun2\_of}\ x\ n = (s,\ t) \longrightarrow |s| = n \wedge |t| = n + 1 \wedge \textit{braun}\ s \wedge \textit{braun}\ t$$

11.5.3 Converting a List into a Braun Tree

We improve on function *adds* from Section 11.3 that has running time $\Theta(n \lg n)$ by developing a linear-time function. Given a list of elements $[1, 2, \ldots]$, we can subdivide it into sublists $[1]$, $[2, 3]$, $[4, \ldots, 7]$, … such that the kth sublist contains the elements of level k of the corresponding Braun tree. This is simply because on each level we have the entries whose index has $k + 1$ bits. Thus we need to process the input list in chunks of size 2^k to produce the trees on level k. But we also need to get the order right. To understand how that works, consider the last two levels of the tree in Figure 11.2:



If we rearrange them in increasing order of the root labels



the following pattern emerges: the left subtrees are labeled $[8, \ldots, 11]$, the right subtrees $[12, \ldots, 15]$. Call t_i the tree with root label i. The correct order of subtrees, i.e. t_4, t_6, t_5, t_7, is restored when the three lists $[t_4,\ t_5]$, $[2,\ 3]$ (the labels above) and $[t_6,\ t_7]$ are combined into new trees by going through them simultaneously from left to right, yielding $[\langle t_4,\ 2,\ t_6\rangle,\ \langle t_5,\ 3,\ t_7\rangle]$, the level above.

Abstracting from this example we arrive at the following code. Loosely speaking, *brauns* k xs produces the Braun trees on level k.

```
brauns :: nat ⇒ 'a list ⇒ 'a tree list
brauns k xs
= (if xs = [] then []
   else let ys = take 2^k xs;
            zs = drop 2^k xs;
            ts = brauns (k + 1) zs
        in nodes ts ys (drop 2^k ts))
```

Function *brauns* chops off a chunk ys of size 2^k from the input list and recursively converts the remainder of the list into a list ts of (at most) 2^{k+1} trees. This list is (conceptually) split into *take* 2^k ts and *drop* 2^k ts which are combined with ys by function *nodes* that traverses its three argument lists simultaneously. As a local optimization, we pass all of ts rather than just *take* 2^k ts to *nodes*.

```
nodes :: 'a tree list ⇒ 'a list ⇒ 'a tree list ⇒ 'a tree list
nodes (l # ls) (x # xs) (r # rs) = ⟨l, x, r⟩ # nodes ls xs rs
nodes (l # ls) (x # xs) [] = ⟨l, x, ⟨⟩⟩ # nodes ls xs []
nodes [] (x # xs) (r # rs) = ⟨⟨⟩, x, r⟩ # nodes [] xs rs
nodes [] (x # xs) [] = ⟨⟨⟩, x, ⟨⟩⟩ # nodes [] xs []
nodes _ [] _ = []
```

Because the input list may not have exactly $2^n - 1$ elements, some of the chunks of elements and trees may be shorter than 2^k. To compensate for that, function *nodes* implicitly pads lists of trees at the end with leaves. This padding is the purpose of equations two to four.

The top-level function for turning a list into a tree simply extracts the first (and only) element from the list computed by *brauns* 0:

```
brauns1 :: 'a list ⇒ 'a tree
brauns1 xs = (if xs = [] then ⟨⟩ else brauns 0 xs ! 0)
```

11.5.3.1 Correctness

The key correctness lemma below expresses a property of Braun trees: the subtrees on level k consist of all elements of the input list xs that are 2^k elements apart, starting from some offset. To state this concisely we define

```
take_nths :: nat ⇒ nat ⇒ 'a list ⇒ 'a list
take_nths _ _ [] = []
take_nths i k (x # xs)
= (if i = 0 then x # take_nths (2^k − 1) k xs else take_nths (i − 1) k xs)
```

The result of *take_nths* i k xs is every 2^k-th element in *drop* i xs.

A number of simple properties follow by easy inductions:

$$take\_nths\ i\ k\ (drop\ j\ xs) = take\_nths\ (i + j)\ k\ xs \tag{11.7}$$

$$\mathit{take\_nths}\ 0\ 0\ xs = xs \tag{11.8}$$

$$\mathit{splice}\ (\mathit{take\_nths}\ 0\ 1\ xs)\ (\mathit{take\_nths}\ 1\ 1\ xs) = xs \tag{11.9}$$

$$\begin{aligned}&\mathit{take\_nths}\ i\ m\ (\mathit{take\_nths}\ j\ n\ xs)\\ &= \mathit{take\_nths}\ (i \cdot 2^n + j)\ (m + n)\ xs\end{aligned} \tag{11.10}$$

$$\mathit{take\_nths}\ i\ k\ xs = [] \longleftrightarrow |xs| \le i \tag{11.11}$$

$$i < |xs| \longrightarrow \mathit{hd}\ (\mathit{take\_nths}\ i\ k\ xs) = xs\ !\ i \tag{11.12}$$

$$\begin{aligned}&|xs| = |ys| \lor |xs| = |ys| + 1 \longrightarrow\\ &\mathit{take\_nths}\ 0\ 1\ (\mathit{splice}\ xs\ ys) = xs \land\\ &\mathit{take\_nths}\ 1\ 1\ (\mathit{splice}\ xs\ ys) = ys\end{aligned} \tag{11.13}$$

$$\begin{aligned}&|\mathit{take\_nths}\ 0\ 1\ xs| = |\mathit{take\_nths}\ 1\ 1\ xs| \lor\\ &|\mathit{take\_nths}\ 0\ 1\ xs| = |\mathit{take\_nths}\ 1\ 1\ xs| + 1\end{aligned} \tag{11.14}$$

We also introduce a predicate relating a tree to a list:

$$\mathit{braun\_list} :: \ 'a\ \mathit{tree} \Rightarrow \ 'a\ \mathit{list} \Rightarrow \mathit{bool}$$

$$\begin{aligned}&\mathit{braun\_list}\ \langle\rangle\ xs = (xs = [])\\ &\mathit{braun\_list}\ \langle l, x, r\rangle\ xs\\ &= (xs \ne [] \land x = \mathit{hd}\ xs \land\\ &\quad \mathit{braun\_list}\ l\ (\mathit{take\_nths}\ 1\ 1\ xs) \land\\ &\quad \mathit{braun\_list}\ r\ (\mathit{take\_nths}\ 2\ 1\ xs))\end{aligned}$$

This definition may look a bit mysterious at first but it satisfies a simple specification: $\mathit{braun\_list}\ t\ xs \longleftrightarrow \mathit{braun}\ t \land xs = \mathit{list}\ t$ (see below). The idea of the above definition is that instead of relating $\langle l, x, r\rangle$ to xs via *splice* we invert the process and relate l and r to the even and odd numbered elements of $\mathit{drop}\ 1\ xs$.

Lemma 11.4. $\mathit{braun\_list}\ t\ xs \longleftrightarrow \mathit{braun}\ t \land xs = \mathit{list}\ t$

Proof by induction on t. The base case is trivial. In the induction step the key properties are (11.14) to prove $\mathit{braun}\ t$ and (11.9) and (11.13) to prove $xs = \mathit{list}\ t$. □

The correctness proof of *brauns* rests on a few simple inductive properties:

$$|\mathit{nodes}\ ls\ xs\ rs| = |xs| \tag{11.15}$$

$$\begin{aligned}&i < |xs| \longrightarrow\\ &\mathit{nodes}\ ls\ xs\ rs\ !\ i\\ &= \langle \textbf{if}\ i < |ls|\ \textbf{then}\ ls\ !\ i\ \textbf{else}\ \langle\rangle, xs\ !\ i,\\ &\quad \textbf{if}\ i < |rs|\ \textbf{then}\ rs\ !\ i\ \textbf{else}\ \langle\rangle\rangle\end{aligned} \tag{11.16}$$

$$|\mathit{brauns}\ k\ xs| = \mathit{min}\ |xs|\ 2^k \tag{11.17}$$

The main theorem expresses the following correctness property of the elements of $\mathit{brauns}\ k\ xs$: every tree $\mathit{brauns}\ k\ xs\ !\ i$ is a Braun tree and its list of elements is $\mathit{take\_nths}\ i\ k\ xs$:

Theorem 11.5. $i < \mathit{min}\ |xs|\ 2^k \longrightarrow \mathit{braun\_list}\ (\mathit{brauns}\ k\ xs\ !\ i)\ (\mathit{take\_nths}\ i\ k\ xs)$

Proof by induction on $|xs|$. Assume $i < \mathit{min}\ |xs|\ 2^k$, which implies $xs \neq []$. Let $zs = \mathit{drop}\ 2^k\ xs$. Thus $|zs| < |xs|$ and therefore the IH applies to zs and yields

$$\begin{aligned}&\forall\, i\ j.\ j = i + 2^k \wedge i < \mathit{min}\ |zs|\ 2^{k+1} \longrightarrow\\ &\qquad \mathit{braun\_list}\ (ts\ !\ i)\ (\mathit{take\_nths}\ j\ (k+1)\ xs)\end{aligned} \tag{$*$}$$

where $ts = \mathit{brauns}\ (k+1)\ zs$. Let $ts' = \mathit{drop}\ 2^k\ ts$. Below we examine $\mathit{nodes}\ ts\ \_\ ts'\ !\ i$ with the help of (11.16). Thus there are four similar cases of which we only discuss one representative one: assume $i < |ts|$ and $i \geq |ts'|$.

$$\begin{aligned}&\mathit{braun\_list}\ (\mathit{brauns}\ k\ xs\ !\ i)\ (\mathit{take\_nths}\ i\ k\ xs)\\ &\longleftrightarrow \mathit{braun\_list}\ (\mathit{nodes}\ ts\ (\mathit{take}\ 2^k\ xs)\ ts'\ !\ i)\ (\mathit{take\_nths}\ i\ k\ xs)\\ &\longleftrightarrow \mathit{braun\_list}\ (ts\ !\ i)\ (\mathit{take\_nths}\ (2^k + i)\ (k+1)\ xs) \wedge\\ &\qquad \mathit{braun\_list}\ \langle\rangle\ (\mathit{take\_nths}\ (2^{k+1} + i)\ (k+1)\ xs)\\ &\qquad\qquad\qquad \text{by (11.16), (11.10), (11.11), (11.12) and assumptions}\\ &\longleftrightarrow \mathit{True} \qquad\qquad \text{by ($*$), (11.11), (11.17) and assumptions}\end{aligned}$$

□

Setting $i = k = 0$ in this theorem we obtain the correctness of $\mathit{brauns1}$ using Lemma 11.4 and (11.8):

Corollary 11.6. $\mathit{braun}\ (\mathit{brauns1}\ xs) \wedge \mathit{list}\ (\mathit{brauns1}\ xs) = xs$

11.5.3.2 Running Time

Function T_{nodes} is shown in Appendix B.4. It is obviously linear:

$$T_{\mathit{nodes}}\ ls\ xs\ rs = |xs| + 1 \tag{11.18}$$

Function T_{brauns} assumes that 2^k can be computed in constant (i.e. 0) time like all basic arithmetic operations. This is justified if k is bounded, in which case 2^k can be implemented as a table lookup.

```
$T_{brauns}$ :: $nat \Rightarrow 'a\ list \Rightarrow nat$
$T_{brauns}\ k\ xs$
= (if $xs = []$ then $0$
   else let $ys = take\ 2^k\ xs$; $zs = drop\ 2^k\ xs$; $ts = brauns\ (k + 1)\ zs$
        in $T_{take}\ 2^k\ xs + T_{drop}\ 2^k\ xs + T_{brauns}\ (k + 1)\ zs + T_{drop}\ 2^k\ ts +$
           $T_{nodes}\ ts\ ys\ (drop\ 2^k\ ts)) + 1$
```

Function T_{brauns} is also linear:

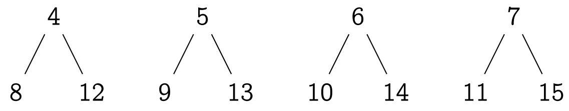
Lemma 11.7. $T_{brauns}\ k\ xs \le 9 \cdot (|xs| + 1)$

Proof by induction on $|xs|$. If $xs = []$ the claim is trivial. Now assume $xs \neq []$ and let $zs = \mathit{drop}\ 2^k\ xs$. In the first step we simplify the body using (11.17), (11.18) and simple properties of *take*, *drop*, T_{take} and T_{drop} and *min*:

$$\begin{aligned}
& T_{brauns}\ k\ xs \\
&= 3 \cdot (\mathit{min}\ 2^k\ |xs| + 1) + (\mathit{min}\ 2^k\ (|xs| - 2^k) + 1) + T_{brauns}\ (k + 1)\ zs + 1 \\
&\le 4 \cdot \mathit{min}\ 2^k\ |xs| + T_{brauns}\ (k + 1)\ zs + 5 \\
&= 4 \cdot \mathit{min}\ 2^k\ |xs| + 9 \cdot (|zs| + 1) + 5 && \text{by IH} \\
&= 4 \cdot \mathit{min}\ 2^k\ |xs| + 9 \cdot (|xs| - 2^k + 1) + 5 \\
&= 4 \cdot \mathit{min}\ 2^k\ |xs| + 4 \cdot (|xs| - 2^k) + 5 \cdot (|xs| - 2^k + 1) + 9 \\
&= 4 \cdot |xs| + 5 \cdot (|xs| - 2^k + 1) + 9 \\
&\le 4 \cdot |xs| + 5 \cdot |xs| + 9 && \text{because } |xs| - 2^k + 1 \le |xs| \\
&= 9 \cdot (|xs| + 1) && \square
\end{aligned}$$

11.5.4 Converting a Braun Tree into a List

We improve on function *list* that has running time $O(n \lg n)$ by developing a linear-time version. Imagine that we want to invert the computation of *brauns1* and thus of *brauns*. Thus it is natural to convert not merely a single tree but a list of trees. Looking once more at the reordered list of subtrees



the following strategy strongly suggests itself: first the roots, then the left children, then the right children. The recursive application of this strategy also takes care of the required reordering of the subtrees. Of course we have to ignore any leaves we encounter. This is the resulting function:

```
list_fast_rec :: 'a tree list ⇒ 'a list
list_fast_rec ts
= (let us = filter (λt. t ≠ ⟨⟩) ts
   in if us = [] then []
      else map value us @ list_fast_rec (map left us @ map right us))
```

$\textit{value}\ \langle l,\ x,\ r\rangle = x$
$\textit{left}\ \langle l,\ x,\ r\rangle = l$
$\textit{right}\ \langle l,\ x,\ r\rangle = r$

Function *list_fast_rec* terminates because *left* and *right* remove the top node of a non-$\langle\rangle$ tree. Thus the sum of the sizes of all trees in ts decreases with each recursive call because us is a non-empty list of non-$\langle\rangle$ trees.

This is the top level function to extract a list from a single tree:

$\textit{list\_fast} :: {}'a\ tree \Rightarrow {}'a\ list$
$\textit{list\_fast}\ t = \textit{list\_fast\_rec}\ [t]$

From *list_fast* one can easily derive an efficient fold function on Braun trees that processes the elements in the tree in the order of their indexes.

11.5.4.1 Correctness

We want to prove correctness of *list_fast*: $\textit{list\_fast}\ t = \textit{list}\ t$ if $\textit{braun}\ t$. A direct proof of $\textit{list\_fast\_rec}\ [t] = \textit{list}\ t$ will fail and we need to generalize this statement to all lists of length 2^k. Reusing the infrastructure from the previous subsection this can be expressed as follows:

Theorem 11.8. $|ts| = 2^k \wedge (\forall i{<}2^k.\ \textit{braun\_list}\ (ts\ !\ i)\ (\textit{take\_nths}\ i\ k\ xs)) \longrightarrow \textit{list\_fast\_rec}\ ts = xs$

Proof by induction on $|xs|$. Assume the two premises. There are two cases.

First assume $|xs| < 2^k$. Then

$$ts = \textit{map}\ (\lambda x.\ \langle\langle\rangle,\ x,\ \langle\rangle\rangle)\ xs\ @\ \textit{replicate}\ n\ \langle\rangle \qquad (*)$$

where $n = |ts| - |xs|$. This can be proved pointwise. Take some $i < 2^k$. If $i < |xs|$ then $\textit{take\_nths}\ i\ k\ xs = \textit{take}\ 1\ (\textit{drop}\ i\ xs)$ (which can be proved by induction on xs). By definition of *braun_list* it follows that $t\ !\ i = \langle l,\ xs\ !\ i,\ r\rangle$ for some l and r such that $\textit{braun\_list}\ l\ []$ and $\textit{braun\_list}\ l\ []$ and thus $l = r = \langle\rangle$, i.e. $t\ !\ i = \langle\langle\rangle,\ xs\ !\ i,\ \langle\rangle\rangle$. If $\neg\ i < |xs|$ then $\textit{take\_nths}\ i\ k\ xs = []$ by (11.11) and thus $\textit{braun\_list}\ (ts\ !\ i)\ []$ by the second premise and thus $ts\ !\ i = \langle\rangle$ by definition of *braun_list*. This concludes the proof of $(*)$. The desired $\textit{list\_fast\_rec}\ ts = xs$ follows easily by definition of *list_fast_rec*.

Now assume $\neg\ |xs| < 2^k$. Then for all $i < 2^k$

$$\begin{array}{l}
ts\ !\ i \neq \langle\rangle \wedge \mathit{value}\ (ts\ !\ i) = xs\ !\ i\ \wedge \\
\mathit{braun\_list}\ (\mathit{left}\ (ts\ !\ i))\ (\mathit{take\_nths}\ (i + 2^k)\ (k + 1)\ xs) \wedge \\
\mathit{braun\_list}\ (\mathit{right}\ (ts\ !\ i))\ (\mathit{take\_nths}\ (i + 2 \cdot 2^k)\ (k + 1)\ xs)
\end{array}$$

follows from the second premise with the help of (11.10), (11.11) and (11.12). We obtain two consequences:

$$\mathit{map\ value}\ ts = \mathit{take}\ 2^k\ xs$$

$$\mathit{list\_fast\_rec}\ (\mathit{map\ left}\ ts\ @\ \mathit{map\ right}\ ts) = \mathit{drop}\ 2^k\ xs$$

The first consequence follows by pointwise reasoning, the second consequence with the help of the IH and (11.7). From these two consequences the desired conclusion $\mathit{list\_fast\_rec}\ ts = xs$ follows by definition of $\mathit{list\_fast\_rec}$. □

11.5.4.2 Running Time

We focus on $\mathit{list\_fast\_rec}$. After a few simplifications with basic properties of map and T_{append}, the definition of $T_{\mathit{list\_fast\_rec}}$ looks like this:

$$\begin{array}{l}
T_{\mathit{list\_fast\_rec}} :: {}'a\ \mathit{tree\ list} \Rightarrow \mathit{nat} \\
T_{\mathit{list\_fast\_rec}}\ ts \\
= (\mathbf{let}\ us = \mathit{filter}\ (\lambda t.\ t \neq \langle\rangle)\ ts \\
\quad\ \mathbf{in}\ |ts| + 1 + \\
\qquad (\mathbf{if}\ us = []\ \mathbf{then}\ 0 \\
\qquad\ \ \mathbf{else}\ 5 \cdot (|us| + 1) + T_{\mathit{list\_fast\_rec}}\ (\mathit{map\ left}\ us\ @\ \mathit{map\ right}\ us))) + 1
\end{array}$$

The following inductive proposition is an abstraction of the core of the termination argument of $\mathit{list\_fast\_rec}$ above.

$$\begin{array}{l}
(\forall t \in \mathit{set}\ ts.\ t \neq \langle\rangle) \longrightarrow \\
(\sum_{t \leftarrow ts} k \cdot |t|) = (\sum_{t \leftarrow \mathit{map\ left}\ ts\ @\ \mathit{map\ right}\ ts} k \cdot |t|) + k \cdot |ts|
\end{array} \tag{11.19}$$

The suggestive notation $\sum x {\leftarrow} xs.\ f\ x$ abbreviates $\mathit{sum\_list}\ (\mathit{map}\ f\ xs)$.

Now we can state and prove a linear upper bound of $T_{\mathit{list\_fast\_rec}}$:

Theorem 11.9. $T_{\mathit{list\_fast\_rec}}\ ts \leq (\sum_{t \leftarrow ts} 14 \cdot |t| + 1) + 2$

Proof by induction on the size of ts (which decreases with each recursive call as we argued above). If $us = []$ the claim is easily seen to be true. Now assume $us \neq []$ and let $\mathit{children} = \mathit{map\ left}\ us\ @\ \mathit{map\ right}\ us$.

$$\begin{array}{ll}
T_{\mathit{list\_fast\_rec}}\ ts = T_{\mathit{list\_fast\_rec}}\ \mathit{children} + 5 \cdot |us| + |ts| + 7 & \\
\leq (\sum_{t \leftarrow \mathit{children}} 14 \cdot |t| + 1) + 5 \cdot |us| + |ts| + 9 & \text{by IH} \\
= (\sum_{t \leftarrow \mathit{children}} 14 \cdot |t|) + 7 \cdot |us| + |ts| + 9 & \\
= (\sum_{t \leftarrow \mathit{children}} 14 \cdot |t|) + 14 \cdot |us| + |ts| + 2 & \text{because } us \neq []
\end{array}$$

$$\begin{aligned}
&= (\textstyle\sum_{t\leftarrow us} 14 \cdot |t|) + |ts| + 2 && \text{by (11.19)}\\
&\le (\textstyle\sum_{t\leftarrow ts} 14 \cdot |t|) + |ts| + 2\\
&= (\textstyle\sum_{t\leftarrow ts} 14 \cdot |t| + 1) + 2 && \square
\end{aligned}$$

11.6 Exercises

Exercise 11.1. Instead of first showing that Braun trees are almost complete, give a direct proof of $\mathit{braun}\ t \longrightarrow h\ t = \lceil \mathit{lg}\ |t|_1 \rceil$ by first showing $\mathit{braun}\ t \longrightarrow 2^{h\ t} \le 2 \cdot |t| + 1$ by induction.

Exercise 11.2. Let *lh*, the “left height”, compute the length of the left spine of a tree. Prove that the left height of a Braun tree is equal to its height: $\mathit{braun}\ t \longrightarrow \mathit{lh}\ t = h\ t$

Exercise 11.3. Give a readable proof of the fact that Braun trees satisfy the same height as size property:

$$\mathit{braun}\ \langle l, x, r\rangle \longrightarrow h\ l = h\ r \lor h\ l = h\ r + 1$$

Hint: use the fact that Braun trees are almost complete (and thus height optimal).

Exercise 11.4. Show that function *bal* in Section 4.3.1 produces Braun trees:

$$n \le |xs| \land \mathit{bal}\ n\ xs = (t, zs) \longrightarrow \mathit{braun}\ t$$

(Isabelle hint: *bal* needs to be qualified as *Balance.bal*.)

Exercise 11.5. One can view Braun trees as tries (see Chapter 12) by indexing them not with a *nat* but a *bool list* where each bit tells us whether to go left or right (as explained at the start of Section 11.2). Function *nat_of* specifies the intended correspondence:

$$\mathit{nat\_of} :: \mathit{bool\ list} \Rightarrow \mathit{nat}$$

$$\begin{aligned}
&\mathit{nat\_of}\ [] = 1\\
&\mathit{nat\_of}\ (b \# bs) = 2 \cdot \mathit{nat\_of}\ bs + (\textbf{if}\ b\ \textbf{then}\ 1\ \textbf{else}\ 0)
\end{aligned}$$

Define the counterparts of *lookup1* and *update1*

$$\begin{aligned}
&\mathit{lookup\_trie} :: 'a\ \mathit{tree} \Rightarrow \mathit{bool\ list} \Rightarrow 'a\\
&\mathit{update\_trie} :: \mathit{bool\ list} \Rightarrow 'a \Rightarrow 'a\ \mathit{tree} \Rightarrow 'a\ \mathit{tree}
\end{aligned}$$

and prove their correctness:

$$\mathit{braun}\ t \land \mathit{nat\_of}\ bs \in \{1..|t|\} \longrightarrow \mathit{lookup\_trie}\ t\ bs = \mathit{lookup1}\ t\ (\mathit{nat\_of}\ bs)$$

$$\mathit{update\_trie}\ bs\ x\ t = \mathit{update1}\ (\mathit{nat\_of}\ bs)\ x\ t$$

Exercise 11.6. Function *del_lo* is defined with the help of function *merge*. Define a recursive function $\mathit{del\_lo2} :: 'a\ \mathit{tree} \Rightarrow 'a\ \mathit{tree}$ without recourse to any auxiliary function and prove $\mathit{del\_lo2}\ t = \mathit{del\_lo}\ t$.

Exercise 11.7. Prove correctness of function *braun_of_naive* defined in Section 11.5.2: $\textit{list}\ (\textit{braun\_of\_naive}\ x\ n) = \textit{replicate}\ n\ x$.

Exercise 11.8. Show that the running time of *size_fast* is quadratic in the height of the tree: Define the running time functions $T_{\textit{diff}}$ and $T_{\textit{size\_fast}}$ (taking 0 time in the base cases) and prove $T_{\textit{size\_fast}}\ t \leq (h\ t)^2$.

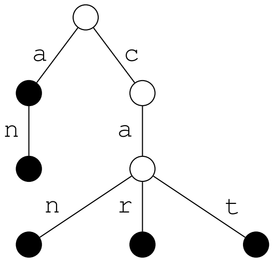
Chapter Notes

Braun trees were investigated by Rem and Braun [1983] and later, in a functional setting, by Hoogerwoord [1992] who coined the term "Braun tree". Section 11.5 is partly based on work by Okasaki [1997]. The whole chapter is based on work by Nipkow and Sewell [2020].

12 Tries

Tobias Nipkow

A **trie** is a search tree where keys are strings, i.e. lists of some type of “characters”. A trie can be viewed as a tree-shaped finite automaton where the root is the start state. For example, the set of strings $\{\mathtt{a}, \mathtt{an}, \mathtt{can}, \mathtt{car}, \mathtt{cat}\}$ is encoded as this trie:



The solid states are accepting, i.e. those nodes terminate the string leading to them.

What distinguishes tries from ordinary search trees is that the access time is not logarithmic in the size of the tree but linear in the length of the string, at least assuming that at each node the transition to the sub-trie takes constant time.

12.1 Abstract Tries via Functions

A nicely abstract model of tries is the following type:

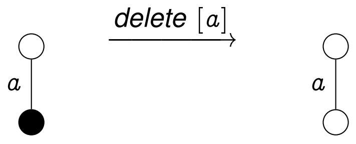
datatype $'a\ trie = \mathsf{Nd}\ bool\ ('a \rightharpoonup 'a\ trie)$

Paremeter $'a$ is the type of “characters”. In a node $\mathsf{Nd}\ b\ f$, b indicates if it is an accepting node and f maps characters to sub-tries. Remember (from Section 6.4) that $\rightharpoonup$ is a type of maps with update notation $f(a \mapsto b)$. There is no *trie* invariant, i.e. the invariant is simply *True*: there are no ordering, balance or other requirements. This is an abstract model that ignores efficiency considerations like fast access to sub-tries.

Figure 12.1 shows how the ADT *Set* is implemented by means of tries. The definitions are straightforward. For simplicity, *delete* does not try to shrink the trie. For example:

```
empty :: 'a trie
empty = Nd False (λ_. None)

isin :: 'a trie ⇒ 'a list ⇒ bool
isin (Nd b _) [] = b
isin (Nd _ m) (k # xs)
= (case m k of None ⇒ False | Some t ⇒ isin t xs)

insert :: 'a list ⇒ 'a trie ⇒ 'a trie
insert [] (Nd _ m) = Nd True m
insert (x # xs) (Nd b m)
= (let s = case m x of None ⇒ empty | Some t ⇒ t
   in Nd b (m(x ↦ insert xs s)))

delete :: 'a list ⇒ 'a trie ⇒ 'a trie
delete [] (Nd _ m) = Nd False m
delete (x # xs) (Nd b m)
= Nd b (case m x of None ⇒ m | Some t ⇒ m(x ↦ delete xs t))
```

Figure 12.1 Implementation of *Set* by tries



Formally:

$$\begin{aligned}&\mathit{delete}\ [a]\ (\mathit{Nd}\ \mathit{False}\ [a \mapsto \mathit{Nd}\ \mathit{True}\ (\lambda\_.\ \mathit{None})])\\&= \mathit{Nd}\ \mathit{False}\ [a \mapsto \mathit{Nd}\ \mathit{False}\ (\lambda\_.\ \mathit{None})]\end{aligned}$$

where $[x \mapsto t] \equiv (\lambda\_.\ \mathit{None})(x \mapsto t)$. The resulting trie is correct (it represents the empty set of strings) but could have been shrunk to *Nd False* $(\lambda\_.\ \mathit{None})$. We will remedy this defect in later, more operational definitions of tries.

12.1.1 Correctness

For the correctness proof we take a lazy approach and define the abstraction function in a trivial manner via *isin*:

$$\mathsf{set\_trie} :: {'a}\ \mathit{trie} \Rightarrow {'a}\ \mathit{list}\ \mathit{set}$$
$$\mathsf{set\_trie}\ t = \{xs \mid \mathsf{isin}\ t\ xs\}$$

Correctness of *empty* and *isin* is trivial, correctness of insertion and deletion is easily proved by induction:

$$\mathsf{set\_trie}\ (\mathsf{insert}\ xs\ t) = \mathsf{set\_trie}\ t \cup \{xs\}$$
$$\mathsf{set\_trie}\ (\mathsf{delete}\ xs\ t) = \mathsf{set\_trie}\ t - \{xs\}$$

This simple model of tries leads to simple correctness proofs but is inefficient because of the function space in ${'a} \rightharpoonup {'a}\ \mathit{trie}$. Now we investigate two efficient implementations: First binary tries where $'a$ is specialized to *bool*. Then ternary tries, where the maps ${'a} \rightharpoonup {'a}\ \mathit{trie}$ are represented by search trees.

12.2 Binary Tries

A **binary trie** is a trie over the alphabet *bool*. That is, binary tries represent sets of *bool lists*. More concretely, a binary trie is simply a binary tree:

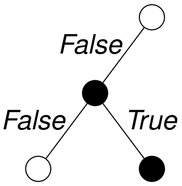
$$\textbf{datatype}\ \mathit{trie} = \mathsf{Lf} \mid \mathsf{Nd}\ \mathit{bool}\ (\mathit{trie} \times \mathit{trie})$$

Grouping the children of a *Nd* together like this is merely for convenience.

A binary trie, for example

$$\mathsf{Nd}\ \mathsf{False}\ (\mathsf{Nd}\ \mathsf{True}\ (\mathsf{Nd}\ \mathsf{False}\ (\mathsf{Lf},\ \mathsf{Lf}),\ \mathsf{Nd}\ \mathsf{True}\ (\mathsf{Lf},\ \mathsf{Lf})),\ \mathsf{Lf})$$

can be visualized like this:



*Lf*s are not shown at all. The edge labels indicated that *False* refers to the left and *True* to the right child. This convention is encoded in the following auxiliary functions selecting from and modifying pairs:

$$\mathsf{sel2} :: \mathit{bool} \Rightarrow {'a} \times {'a} \Rightarrow {'a}$$
$$\mathsf{sel2}\ b\ (a_1,\ a_2) = (\textbf{if}\ b\ \textbf{then}\ a_2\ \textbf{else}\ a_1)$$

```
empty :: trie
empty = Lf

isin :: trie ⇒ bool list ⇒ bool
isin Lf _ = False
isin (Nd b lr) ks = (case ks of [] ⇒ b | k # ks' ⇒ isin (sel2 k lr) ks')

insert :: bool list ⇒ trie ⇒ trie
insert [] Lf = Nd True (Lf, Lf)
insert [] (Nd _ lr) = Nd True lr
insert (k # ks) Lf = Nd False (mod2 (insert ks) k (Lf, Lf))
insert (k # ks) (Nd b lr) = Nd b (mod2 (insert ks) k lr)

delete :: bool list ⇒ trie ⇒ trie
delete _ Lf = Lf
delete ks (Nd b lr)
= (case ks of [] ⇒ node False lr
   | k # ks' ⇒ node b (mod2 (delete ks') k lr))

node b lr = (if ¬ b ∧ lr = (Lf, Lf) then Lf else Nd b lr)
```

Figure 12.2 Implementation of *Set* by binary tries

```
mod2 :: ('a ⇒ 'a) ⇒ bool ⇒ 'a × 'a ⇒ 'a × 'a
mod2 f b (a₁, a₂) = (if b then (a₁, f a₂) else (f a₁, a₂))
```

The implementation of the *Set* interface is shown in Figure 12.2. In our abstract tries, deletion could generate non-empty sub-tries that do not contain an accepting *Nd*. In contrast, our binary *delete* employs a smart constructor *node* that shrinks a non-accepting *Nd* to a *Lf* if both children have become empty. For example *delete* [*True*] (*Nd False* (*Lf*, *Nd True* (*Lf*, *Lf*))) = *Lf*.

To ensure that tries are fully shrunk at all times, we make this constraint an invariant: if both sub-tries of a *Nd* are *Lf*s, the *Nd* must be accepting.

$\mathit{invar} :: \mathit{trie} \Rightarrow \mathit{bool}$

$\mathit{invar}\ \mathit{Lf} = \mathit{True}$
$\mathit{invar}\ (\mathit{Nd}\ b\ (l,\ r)) = (\mathit{invar}\ l \wedge \mathit{invar}\ r \wedge (l = \mathit{Lf} \wedge r = \mathit{Lf} \longrightarrow b))$

Of course we will need to prove that it is invariant.

12.2.1 Correctness

For the correctness proof we take the same lazy approach as above:

$\mathit{set\_trie} :: \mathit{trie} \Rightarrow \mathit{bool\ list\ set}$

$\mathit{set\_trie}\ t = \{xs \mid \mathit{isin}\ t\ xs\}$

The two non-trivial functional correctness properties

$$\mathit{set\_trie}\ (\mathit{insert}\ xs\ t) = \mathit{set\_trie}\ t \cup \{xs\} \tag{12.1}$$

$$\mathit{set\_trie}\ (\mathit{delete}\ xs\ t) = \mathit{set\_trie}\ t - \{xs\} \tag{12.2}$$

are simple consequences of the following inductive properties:

$$\mathit{isin}\ (\mathit{insert}\ xs\ t)\ ys = (xs = ys \vee \mathit{isin}\ t\ ys)$$
$$\mathit{isin}\ (\mathit{delete}\ xs\ t)\ ys = (xs \neq ys \wedge \mathit{isin}\ t\ ys)$$

The invariant is not required because it only expresses a space optimality property.

Preservation of the invariant is easily proved by induction:

$$\mathit{invar}\ t \longrightarrow \mathit{invar}\ (\mathit{insert}\ xs\ t)$$
$$\mathit{invar}\ t \longrightarrow \mathit{invar}\ (\mathit{delete}\ xs\ t)$$

12.2.2 Exercises

Exercise 12.1. Show that distinct tries (which satisfy invar) represent distinct sets:

$$\mathit{invar}\ t_1 \wedge \mathit{invar}\ t_2 \longrightarrow (\mathit{set\_trie}\ t_1 = \mathit{set\_trie}\ t_2) = (t_1 = t_2)$$

This is in contrast with most BST representations of sets.

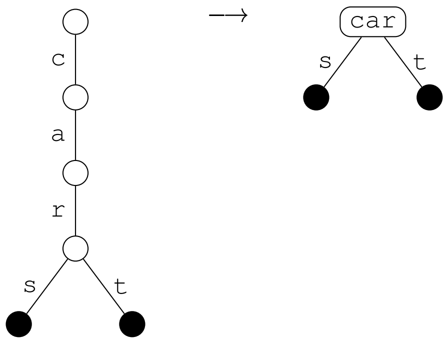
Exercise 12.2. Define a union operation $\mathit{union} :: \mathit{trie} \Rightarrow \mathit{trie} \Rightarrow \mathit{trie}$ on binary tries and prove $\mathit{set\_trie}\ (\mathit{union}\ t_1\ t_2) = \mathit{set\_trie}\ t_1 \cup \mathit{set\_trie}\ t_2$ and $\mathit{invar}\ t_1 \wedge \mathit{invar}\ t_2 \longrightarrow \mathit{invar}\ (\mathit{union}\ t_1\ t_2)$. Similarly for intersection where you should be able to prove $\mathit{invar}\ (\mathit{inter}\ t_1\ t_2)$ outright.

Exercise 12.3. This exercise is about searching tries with wildcard patterns, i.e. strings that can contain a special symbol that matches any character. We model such patterns with type *bool option list* where any Boolean value matches *None* but only b matches *Some* b. Define a function *matches* :: *'a option list* ⇒ *'a list* ⇒ *bool* that expresses when a wildcard pattern is matched by a *bool list*. Then define a function *isins* :: *trie* ⇒ *bool option list* ⇒ *bool list list* that searches a trie with a wildcard pattern and returns all the *bool lists* in the trie that match the pattern. Prove its correctness: $(xs \in \mathit{set}\ (\mathit{isins}\ t\ ps)) = (\mathit{isin}\ t\ xs \wedge \mathit{matches}\ ps\ xs)$.

Exercise 12.4. This exercise is about nearest-neighbour search, namely finding all strings in a trie within a given Hamming distance of the search key. The Hamming distance of two lists of the same length is the number of positions where they differ. Define a function *Hdist* :: *'a list* ⇒ *'a list* ⇒ *nat* that computes the Hamming distance. Then define a function *near* :: *trie* ⇒ *bool list* ⇒ *nat* ⇒ *bool list list* such that *near* t xs d is a list of all ys in t of the same length as xs that have Hamming distance at most d from xs. Prove its correctness:
$(ys \in \mathit{set}\ (\mathit{near}\ t\ xs\ d)) = (|xs| = |ys| \wedge \mathit{isin}\ t\ ys \wedge \mathit{Hdist}\ xs\ ys \le d)$.

12.3 Binary Patricia Tries

Tries can contain long branches without branching. These can be contracted by storing the branch directly in the start node. The result is called a **Patricia trie**. The following figure shows the contraction of a trie into a Patricia trie:



This is the data type of binary Patricia tries:

datatype *trieP* = *LfP* | *NdP* (*bool list*) *bool* (*trieP* × *trieP*)

The implementation of the *Set* ADT by binary Patricia tries is shown in Figure 12.3; function *nodeP* is displayed separately. The key auxiliary function is *lcp* where *lcp* xs ys = (ps, xs', ys') such that ps is the longest common prefix of xs and ys and xs'/ys' is what remains of xs/ys after dropping ps. Function *lcp* is used by both *insertP* and *deleteP* to analyze how the given key and the prefix stored in the *NdP* overlap. For the detailed case analysis see the code.

Just as for basic binary tries, deletion may enable shrinking. For example, *NdP* xs *False* (*NdP* ys b lr, *LfP*) can be shrunk to *NdP* (xs @ *False* # ys) b lr: both tries represent the same set. Function *deleteP* performs shrinking with the help of the smart constructor *nodeP* that merges two nested *NdP*'s if there is no branching:

```
nodeP ps b lr
= (if b then NdP ps b lr
   else case lr of
        (LfP, LfP) ⇒ LfP |
        (LfP, NdP ks b lr) ⇒ NdP (ps @ True # ks) b lr |
        (NdP ks b lr, LfP) ⇒ NdP (ps @ False # ks) b lr |
        _ ⇒ NdP ps b lr)
```

This shrinking property motivates the following invariant: any non-branching *NdP* must be accepting (because otherwise it could be merged with its children).

```
invarP :: trieP ⇒ bool
invarP LfP = True
invarP (NdP _ b (l, r)) = (invarP l ∧ invarP r ∧ (l = LfP ∨ r = LfP ⟶ b))
```

It is tempting to think that *invarP* t = *invar* (*abs_trieP* t) but this is not the case. Find a t such that ¬ *invarP* t but *invar* (*abs_trieP* t).

12.3.1 Correctness

This is an exercise in stepwise data refinement. We have already proved that *trie* implements *Set* via an abstraction function. Now we map *trieP* back to *trie* via another abstraction function. Afterwards the overall correctness follows trivially by composing the two abstraction functions.

The abstraction function *abs_trieP* is defined via the auxiliary function *prefix_trie* that prefixes a trie with a bit list:

```
abs_trieP :: trieP ⇒ trie
abs_trieP LfP = Lf
abs_trieP (NdP ps b (l, r)) = prefix_trie ps (Nd b (abs_trieP l, abs_trieP r))
```

```
emptyP :: trieP
emptyP = LfP

isinP :: trieP ⇒ bool list ⇒ bool
isinP LfP _ = False
isinP (NdP ps b lr) ks
= (let n = |ps|
   in if ps = take n ks then case drop n ks of
                                  [] ⇒ b |
                                  k # x ⇒ isinP (sel2 k lr) x
      else False)

insertP :: bool list ⇒ trieP ⇒ trieP
insertP ks LfP = NdP ks True (LfP, LfP)
insertP ks (NdP ps b lr)
= (case lcp ks ps of
   (_, [], []) ⇒ NdP ps True lr |
   (qs, [], p # ps') ⇒
     let t = NdP ps' b lr
     in NdP qs True (if p then (LfP, t) else (t, LfP)) |
   (_, k # ks', []) ⇒ NdP ps b (mod2 (insertP ks') k lr) |
   (qs, k # ks', _ # ps') ⇒
     let tp = NdP ps' b lr; tk = NdP ks' True (LfP, LfP)
     in NdP qs False (if k then (tp, tk) else (tk, tp)))

deleteP :: bool list ⇒ trieP ⇒ trieP
deleteP ks LfP = LfP
deleteP ks (NdP ps b lr)
= (case lcp ks ps of
    (_, [], []) ⇒ nodeP ps False lr |
    (_, _, _ # _) ⇒ NdP ps b lr |
    (_, k # ks', []) ⇒ nodeP ps b (mod2 (deleteP ks') k lr))

lcp :: 'a list ⇒ 'a list ⇒ 'a list × 'a list × 'a list
lcp [] ys = ([], [], ys)
lcp xs [] = ([], xs, [])
lcp (x # xs) (y # ys)
= (if x ≠ y then ([], x # xs, y # ys)
   else let (ps, xs', ys') = lcp xs ys in (x # ps, xs', ys'))
```

Figure 12.3 Implementation of *Set* by binary Patricia tries

$$\mathit{prefix\_trie} :: \mathit{bool\ list} \Rightarrow \mathit{trie} \Rightarrow \mathit{trie}$$
$$\mathit{prefix\_trie}\ []\ t = t$$
$$\mathit{prefix\_trie}\ (k\ \#\ \mathit{ks})\ t$$
$$= (\mathbf{let}\ t' = \mathit{prefix\_trie}\ \mathit{ks}\ t\ \mathbf{in}\ \mathit{Nd}\ \mathit{False}\ (\mathbf{if}\ k\ \mathbf{then}\ (\mathit{Lf},\ t')\ \mathbf{else}\ (t',\ \mathit{Lf})))$$

Correctness of *emptyP* is trivial. Correctness of the remaining operations is proved by induction and requires a number of supporting inductive lemmas which we display before the corresponding correctness properties.

Correctness of *isinP*:

$$\mathit{isin}\ (\mathit{prefix\_trie}\ \mathit{ps}\ t)\ \mathit{ks} = (\mathit{ps} = \mathit{take}\ |\mathit{ps}|\ \mathit{ks} \wedge \mathit{isin}\ t\ (\mathit{drop}\ |\mathit{ps}|\ \mathit{ks}))$$
$$\mathit{isinP}\ t\ \mathit{ks} = \mathit{isin}\ (\mathit{abs\_trieP}\ t)\ \mathit{ks}$$

Correctness of *insertP*:

$$\mathit{prefix\_trie}\ \mathit{ks}\ (\mathit{Nd}\ \mathit{True}\ (\mathit{Lf},\ \mathit{Lf})) = \mathit{insert}\ \mathit{ks}\ \mathit{Lf}$$
$$\mathit{insert}\ \mathit{ps}\ (\mathit{prefix\_trie}\ \mathit{ps}\ (\mathit{Nd}\ b\ \mathit{lr})) = \mathit{prefix\_trie}\ \mathit{ps}\ (\mathit{Nd}\ \mathit{True}\ \mathit{lr})$$
$$\mathit{insert}\ (\mathit{ks}\ @\ \mathit{ks}')\ (\mathit{prefix\_trie}\ \mathit{ks}\ t) = \mathit{prefix\_trie}\ \mathit{ks}\ (\mathit{insert}\ \mathit{ks}'\ t)$$
$$\mathit{prefix\_trie}\ (\mathit{ps}\ @\ \mathit{qs})\ t = \mathit{prefix\_trie}\ \mathit{ps}\ (\mathit{prefix\_trie}\ \mathit{qs}\ t)$$
$$\mathit{lcp}\ \mathit{ks}\ \mathit{ps} = (\mathit{qs},\ \mathit{ks}',\ \mathit{ps}') \longrightarrow$$
$$\mathit{ks} = \mathit{qs}\ @\ \mathit{ks}' \wedge \mathit{ps} = \mathit{qs}\ @\ \mathit{ps}' \wedge (\mathit{ks}' \neq [] \wedge \mathit{ps}' \neq [] \longrightarrow \mathit{hd}\ \mathit{ks}' \neq \mathit{hd}\ \mathit{ps}')$$
$$\mathit{abs\_trieP}\ (\mathit{insertP}\ \mathit{ks}\ t) = \mathit{insert}\ \mathit{ks}\ (\mathit{abs\_trieP}\ t) \tag{12.3}$$
$$\mathit{invarP}\ t \longrightarrow \mathit{invarP}\ (\mathit{insertP}\ \mathit{xs}\ t)$$

Correctness of *deleteP*:

$$\mathit{delete}\ \mathit{xs}\ (\mathit{prefix\_trie}\ \mathit{xs}\ (\mathit{Nd}\ b\ (l,\ r)))$$
$$= (\mathbf{if}\ (l,\ r) = (\mathit{Lf},\ \mathit{Lf})\ \mathbf{then}\ \mathit{Lf}\ \mathbf{else}\ \mathit{prefix\_trie}\ \mathit{xs}\ (\mathit{Nd}\ \mathit{False}\ (l,\ r)))$$
$$\mathit{delete}\ (\mathit{xs}\ @\ \mathit{ys})\ (\mathit{prefix\_trie}\ \mathit{xs}\ t)$$
$$= (\mathbf{if}\ \mathit{delete}\ \mathit{ys}\ t = \mathit{Lf}\ \mathbf{then}\ \mathit{Lf}\ \mathbf{else}\ \mathit{prefix\_trie}\ \mathit{xs}\ (\mathit{delete}\ \mathit{ys}\ t))$$
$$\mathit{abs\_trieP}\ (\mathit{deleteP}\ \mathit{ks}\ t) = \mathit{delete}\ \mathit{ks}\ (\mathit{abs\_trieP}\ t) \tag{12.4}$$
$$\mathit{invarP}\ t \longrightarrow \mathit{invarP}\ (\mathit{deleteP}\ \mathit{xs}\ t)$$

It is now trivial to obtain the correctness of the *trieP* implementation of sets. The invariant is still *invarP* and has already been dealt with. The abstraction function is simply the composition of the two abstraction functions: $\mathit{set\_trieP} = \mathit{set\_trie} \circ \mathit{abs\_trieP}$. The required functional correctness properties (ignoring *emptyP* and *isinP*) are trivial compositions of (12.1)/(12.2) and (12.3)/(12.4):

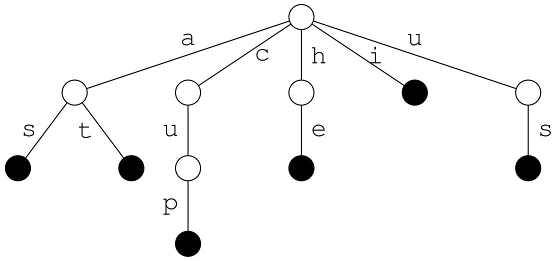
$$\mathit{set\_trieP}\ (\mathit{insertP}\ \mathit{xs}\ t) = \mathit{set\_trieP}\ t \cup \{\mathit{xs}\}$$
$$\mathit{set\_trieP}\ (\mathit{deleteP}\ \mathit{xs}\ t) = \mathit{set\_trieP}\ t - \{\mathit{xs}\}$$

12.3.2 Exercises

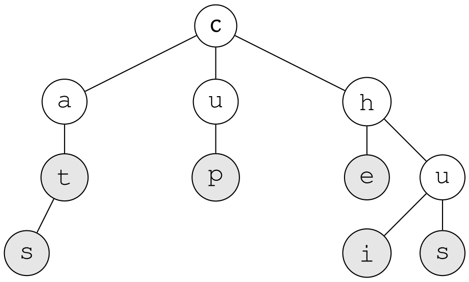
The exercises for binary tries (Section 12.2.2) can be repeated for binary Patricia tries.

12.4 Ternary Tries

What if we want to implement our original abstract tries over type $'a$ efficiently, not just binary tries? For example the following one:



Ternary tries implement the $'a \rightharpoonup 'a\ trie$ maps as BSTs. The above trie can be represented (non-uniquely) by the following ternary trie:



The ternary trie diagram should be interpreted as follows. The left and right children of a node form the BST. The middle child is the sub-trie that the character in the node maps to. Accepting nodes are gray. The name **ternary trie** derives from the fact that nodes have three children. However, conceptually they are BSTs that map elements of type $'a$ to further such BSTs, i.e. the middle child isn't really a child but part of the contents of the node.

Using the unbalanced tree implementation of maps from Section 6.5 (any other map implementation works just as well) we define ternary tries as follows:

datatype $'a\ trie3 = \mathit{Nd3}\ bool\ (('a \times 'a\ trie3)\ tree)$

As before, the *bool* field indicates if it is an accepting node.

The invariant for ternary tries requires that in all nodes the invariant *invar* of the map implementation holds:

```
empty3 :: 'a trie3
empty3 = Nd3 False ⟨⟩

isin3 :: 'a trie3 ⇒ 'a list ⇒ bool
isin3 (Nd3 b _) [] = b
isin3 (Nd3 _ m) (x # xs)
= (case lookup m x of None ⇒ False | Some t ⇒ isin3 t xs)

insert3 :: 'a list ⇒ 'a trie3 ⇒ 'a trie3
insert3 [] (Nd3 _ m) = Nd3 True m
insert3 (x # xs) (Nd3 b m)
= Nd3 b
    (update x
      (insert3 xs (case lookup m x of None ⇒ empty3 | Some t ⇒ t)) m)

delete3 :: 'a list ⇒ 'a trie3 ⇒ 'a trie3
delete3 [] (Nd3 _ m) = Nd3 False m
delete3 (x # xs) (Nd3 b m)
= Nd3 b
    (case lookup m x of None ⇒ m | Some t ⇒ update x (delete3 xs t) m)
```

Figure 12.4 Implementation of *Set* via ternary tries

```
invar3 :: 'a trie3 ⇒ bool
invar3 (Nd3 _ m) = (invar m ∧ (∀ a t. lookup m a = Some t ⟶ invar3 t))
```

The self-explanatory implementation of the *Set* interface is shown in Figure 12.4. Function *delete* does not try to shrink the trie. Remember that *lookup* and *update* come from the *Map* implementation.

12.4.1 Correctness

This is another example of stepwise refinement, just like in the correctness proof for binary Patricia tries in Section 12.3. We show that *'a trie3* implements *'a trie* (from Section 12.1) via this abstraction function:

$$\mathit{abs3} :: {}'a\ \mathit{trie3} \Rightarrow {}'a\ \mathit{trie}$$
$$\mathit{abs3}\ (\mathit{Nd3}\ b\ t) = \mathit{Nd}\ b\ (\lambda a.\ \mathit{map\_option}\ \mathit{abs3}\ (\mathit{lookup}\ t\ a))$$

$$\mathit{map\_option} :: ({}'a \Rightarrow {}'b) \Rightarrow {}'a\ \mathit{option} \Rightarrow {}'b\ \mathit{option}$$
$$\mathit{map\_option}\ f\ \mathit{None} = \mathit{None}$$
$$\mathit{map\_option}\ f\ (\mathit{Some}\ x) = \mathit{Some}\ (f\ x)$$

The correctness properties (ignoring *empty3*) have easy inductive proofs:

$$\mathit{isin3}\ t\ xs = \mathit{isin}\ (\mathit{abs3}\ t)\ xs$$
$$\mathit{invar3}\ t \longrightarrow \mathit{abs3}\ (\mathit{insert3}\ xs\ t) = \mathit{insert}\ xs\ (\mathit{abs3}\ t)$$
$$\mathit{invar3}\ t \longrightarrow \mathit{abs3}\ (\mathit{delete3}\ xs\ t) = \mathit{delete}\ xs\ (\mathit{abs3}\ t)$$
$$\mathit{invar3}\ t \longrightarrow \mathit{invar3}\ (\mathit{insert3}\ xs\ t)$$
$$\mathit{invar3}\ t \longrightarrow \mathit{invar3}\ (\mathit{delete3}\ xs\ t)$$

We had already shown that ${}'a\ \mathit{trie}$ implements ${}'a\ \mathit{set}$ and composing the abstraction functions and correctness theorems to show that ${}'a\ \mathit{trie3}$ implements ${}'a\ \mathit{set}$ is trivial.

Chapter Notes

Tries were first sketched by De La Briandais [1959] and described in more detail by Fredkin [1960] who coined their name based on the word reTRIEval. However, “trie” is usually pronounced like “try” rather than “tree” to avoid confusion. Patricia tries are due to Morrison [1968]. Ternary tries are due to Bentley and Sedgewick [1997].

Appel and Leroy [2023] present verified binary tries with an emphasis on efficiency.

13 Region Quadtrees

Tobias Nipkow

Quadtrees are a well-known data structure for the hierarchical representation of two-dimensional space in computer graphics, image processing, computational geometry, geographic information systems, and related areas. There are many variants of quadtrees and we concentrate on **region quadtrees**. They are particularly well suited to the representation of two-dimensional images of pixels because of a potentially significant compression of the image. As all hierarchical data structures, they support parallel processing naturally. We consider the following variants:

- Basic region quadtrees (Section 13.1)
- Representation of block matrices via region quadtrees (Section 13.2)
- Region quadtrees generalized from two to k dimensions (Section 13.3)

In each case we verify a small selection of representative operations.

13.1 Region Quadtrees

The best-known form of region quadtrees represent two-dimensional images of **pixels** that can be black or white. The image is recursively subdivided into four quadrants until all pixels in a quadrant have the same value. Consequently the image must be of size $2^n \times 2^n$ pixels. The number n is called the **resolution** of the quadtree. The quadrants are numbered like this:

$$\begin{array}{|c|c|}\hline 1 & 3 \\ \hline 0 & 2 \\ \hline \end{array} \tag{13.1}$$

An image and its quadtree representation is shown in Figure 13.1. The gray nodes in the tree represent subdivided squares.

The representation of quadtrees as a data type

```
datatype 'a qtree = L 'a | Q ('a qtree) ('a qtree) ('a qtree) ('a qtree)
```

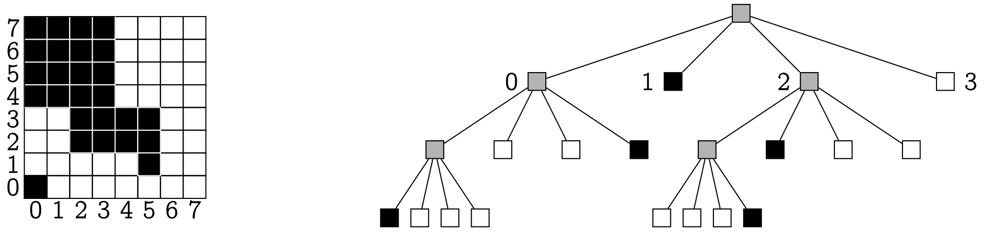


Figure 13.1 Image and corresponding quadtree

supports leaves (constructor L) where all pixels have the same value of the parameter type $'a$. Black and white images as seen in Figure 13.1 are represented by boolean quadtrees, i.e. where $'a = bool$.

The height of a quadtree is defined as usual:

$$\mathit{height} :: 'a\ qtree \Rightarrow nat$$

$$\mathit{height}\ (L\ \_) = 0$$
$$\mathit{height}\ (Q\ t_0\ t_1\ t_2\ t_3) = \mathit{Max}\ \{\mathit{height}\ t_0,\ \mathit{height}\ t_1,\ \mathit{height}\ t_2,\ \mathit{height}\ t_3\} + 1$$

A quadtree is *compressed* if no subtree could be replaced by a leaf:

$$\mathit{compressed} :: 'a\ qtree \Rightarrow bool$$

$$\mathit{compressed}\ (L\ \_) = \mathit{True}$$
$$\mathit{compressed}\ (Q\ t_0\ t_1\ t_2\ t_3)$$
$$= (\mathit{compressed}\ t_0 \wedge \mathit{compressed}\ t_1 \wedge \mathit{compressed}\ t_2 \wedge \mathit{compressed}\ t_3 \wedge$$
$$(\nexists x.\ t_0 = L\ x \wedge t_1 = t_0 \wedge t_2 = t_0 \wedge t_3 = t_0))$$

To keep our quadtrees compressed, we construct them with the compressing constructor Qc, which assumes that its arguments are already compressed:

$$Qc :: 'a\ qtree \Rightarrow 'a\ qtree \Rightarrow 'a\ qtree \Rightarrow 'a\ qtree \Rightarrow 'a\ qtree$$

$$Qc\ (L\ x_0)\ (L\ x_1)\ (L\ x_2)\ (L\ x_3)$$
$$= (\textbf{if}\ x_0 = x_1 \wedge x_1 = x_2 \wedge x_2 = x_3\ \textbf{then}\ L\ x_0$$
$$\quad \textbf{else}\ Q\ (L\ x_0)\ (L\ x_1)\ (L\ x_2)\ (L\ x_3))$$
$$Qc\ t_0\ t_1\ t_2\ t_3 = Q\ t_0\ t_1\ t_2\ t_3$$

The following property of Qc is frequently used:

$$\mathit{compressed}\ t_0 \wedge \mathit{compressed}\ t_1 \wedge \mathit{compressed}\ t_2 \wedge \mathit{compressed}\ t_3 \longrightarrow$$
$$\mathit{compressed}\ (\mathit{Qc}\ t_0\ t_1\ t_2\ t_3)$$

A quadtree does not specify the resolution of the image it represents. For example, *L True* can represent a square of any size $2^n \times 2^n$. One can explicitly pair a quadtree with its resolution, or one can keep both separate, as we will do. Either way, the tree and the resolution have to match, i.e. $\mathit{height}\ t \le n$, which one can see as an invariant of the pair $(t,\ n)$. Otherwise t cannot always represent an image of size $2^n \times 2^n$. For example, *Q* (*L True*) (*L True*) (*L True*) (*L False*) does not represent an image of size 1×1 but requires at least 2×2 pixels. Therefore functions on quadtrees often take the intended resolution n as an argument.

13.1.1 Functions *get* and *put*

Trees of type $'a\ \mathit{qtree}$ can be viewed as representations of mappings from $(i,\ j)$ coordinates to values of type $'a$. Thus the operation *get* for extracting a single pixel doubles as the abstraction function:

$$\mathit{get} :: \mathit{nat} \Rightarrow {'a}\ \mathit{qtree} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat} \Rightarrow {'a}$$

$$\mathit{get}\ \_\ \ (L\ b)\ \_\ \_\ = b$$
$$\mathit{get}\ (n + 1)\ (Q\ t_0\ t_1\ t_2\ t_3)\ i\ j$$
$$= \mathit{get}\ n\ (\mathit{select}\ (i < 2^n)\ (j < 2^n)\ t_0\ t_1\ t_2\ t_3)\ (i \bmod 2^n)\ (j \bmod 2^n)$$

$$\mathit{select} :: \mathit{bool} \Rightarrow \mathit{bool} \Rightarrow {'a} \Rightarrow {'a} \Rightarrow {'a} \Rightarrow {'a} \Rightarrow {'a}$$

$$\mathit{select}\ x\ y\ t_0\ t_1\ t_2\ t_3$$
$$= (\textbf{if}\ x\ \textbf{then if}\ y\ \textbf{then}\ t_0\ \textbf{else}\ t_1\ \textbf{else if}\ y\ \textbf{then}\ t_2\ \textbf{else}\ t_3)$$

The call $\mathit{get}\ n\ t\ i\ j$ returns the pixel at coordinate $(i,\ j)$ from the image of resolution n represented by tree t. Function *select* selects one of four quadrants addressed by two booleans. For an efficient implementation one should replace 2^n by something like a table lookup or work directly with machine words.

Note that $\mathit{get}\ n\ t\ i\ j$ is only defined if $\mathit{height}\ t \le n$. The reason for this was discussed above. Partiality is the norm for functions that take both a quadtree and its resolution. This is reflected in the functions' properties, which are conditional (e.g. the properties of *put* below).

Although *get* does not require $i, j < 2^n$ (they are simply forced into that range via mod 2^n) this natural restriction is sometimes needed. The restriction is conveniently expressed as $(i,\ j) \in \mathit{sq}\ n$ where

$sq\ n = \{(i, j) \mid i < 2^n \wedge j < 2^n\}$

The converse of *get* is *put*, for setting a single pixel:

```
put :: nat ⇒ nat ⇒ 'a ⇒ nat ⇒ 'a qtree ⇒ 'a qtree
put _ _ a 0 (L _) = L a
put i j a (n + 1) t
= modify (put (i mod 2^n) (j mod 2^n) a n) (i < 2^n) (j < 2^n)
    (case t of L b ⇒ (L b, L b, L b, L b) | Q t0 t1 t2 t3 ⇒ (t0, t1, t2, t3))

modify ::
  ('a qtree ⇒ 'a qtree)
  ⇒ bool ⇒ bool ⇒ 'a qtree × 'a qtree × 'a qtree × 'a qtree ⇒ 'a qtree
modify f x y (t0, t1, t2, t3)
= (if x then if y then Qc (f t0) t1 t2 t3 else Qc t0 (f t1) t2 t3
    else if y then Qc t0 t1 (f t2) t3 else Qc t0 t1 t2 (f t3))
```

Note that when recombining quadrants on the way back up, *Q* is replaced by *Qc* to take care of possible compressions.

Correctness is expressed by a triple of properties: functional correctness, preservation of resolution and compression.

$$\mathit{height}\ t \le n \wedge (i, j) \in \mathit{sq}\ n \wedge (i', j') \in \mathit{sq}\ n \longrightarrow$$
$$\mathit{get}\ n\ (\mathit{put}\ i\ j\ a\ n\ t)\ i'\ j' = (\textbf{if}\ i' = i \wedge j' = j\ \textbf{then}\ a\ \textbf{else}\ \mathit{get}\ n\ t\ i'\ j')$$

$$\mathit{height}\ t \le n \longrightarrow \mathit{height}\ (\mathit{put}\ i\ j\ a\ n\ t) \le n$$

$$\mathit{height}\ t \le n \wedge \mathit{compressed}\ t \longrightarrow \mathit{compressed}\ (\mathit{put}\ i\ j\ a\ n\ t)$$

Note that the special case of *bool qtree* can be viewed as a representation of a set of points: $\{(i, j) \mid (i, j) \in \mathit{sq}\ n \wedge \mathit{get}\ n\ t\ i\ j\}$. Function *get* is also the *isin*-test and *put* combines *insert* and *delete*.

There is a wide range of interesting functions on quadtrees. What follows should be considered a not quite random sample from a much larger space.

13.1.2 Boolean Operations

As remarked above, boolean quadtrees represent sets. It turns out that they support binary set operations like ∪, ∩, etc. even more naturally than manipulation of individual pixels. They can be expressed as a simple simultaneous traversal of both trees and basic boolean operations on the leaves. As an example we consider intersection:

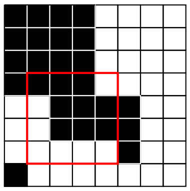


Figure 13.2 Image and subimage

$\mathit{inter} :: \mathit{bool\ qtree} \Rightarrow \mathit{bool\ qtree} \Rightarrow \mathit{bool\ qtree}$

$\mathit{inter}\ (L\ b)\ t = (\mathbf{if}\ b\ \mathbf{then}\ t\ \mathbf{else}\ L\ \mathit{False})$
$\mathit{inter}\ t\ (L\ b) = (\mathbf{if}\ b\ \mathbf{then}\ t\ \mathbf{else}\ L\ \mathit{False})$
$\mathit{inter}\ (Q\ s_1\ s_2\ s_3\ s_4)\ (Q\ t_1\ t_2\ t_3\ t_4)$
$= \mathit{Qc}\ (\mathit{inter}\ s_1\ t_1)\ (\mathit{inter}\ s_2\ t_2)\ (\mathit{inter}\ s_3\ t_3)\ (\mathit{inter}\ s_4\ t_4)$

Other set operations (union, difference, xor) can be defined analogously, with different base cases.

The correctness theorems are easily stated and proved

$$\mathit{height}\ t_1 \le n \wedge \mathit{height}\ t_2 \le n \longrightarrow$$
$$\mathit{get}\ n\ (\mathit{inter}\ t_1\ t_2)\ i\ j = (\mathit{get}\ n\ t_1\ i\ j \wedge \mathit{get}\ n\ t_2\ i\ j)$$

$$\mathit{height}\ (\mathit{inter}\ t_1\ t_2) \le \mathit{max}\ (\mathit{height}\ t_1)\ (\mathit{height}\ t_2)$$

$$\mathit{compressed}\ t_1 \wedge \mathit{compressed}\ t_2 \longrightarrow \mathit{compressed}\ (\mathit{inter}\ t_1\ t_2)$$

Exercise 13.1. Define and verify the operations of set union and set difference on boolean quadtrees.

13.1.3 Extracting Subimages

As an example of a graphics-oriented function consider the extraction of a subimage (a square of size $2^m \times 2^m$) in the form of a new quadtree. Figure 13.2 shows such a subimage with a red border.

Below we define a function $\mathit{get\_sq}\ n\ t\ m\ i\ j$ that takes a quadtree t and its resolution n and extracts a quadtree of the subimage of resolution m with lower left corner at (i, j). It is a bit tricky because it can involve subimages of varying sizes from all four quadrants of a quadtree. Function *get_sq* recurses over t and m as follows. If the subimage is completely within one quadrant, *get_sq* descends into that quadrant (via *select*). Otherwise the subimage needs to be assembled from smaller subimages from multiple quadrants.

$$
\begin{array}{l}
\textsf{get\_sq} :: \mathit{nat} \Rightarrow {'a}\ \mathit{qtree} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat} \Rightarrow {'a}\ \mathit{qtree} \\
\textsf{get\_sq}\ \_\ \ (\textsf{L}\ b)\ \_\ \_\ \_\ = \textsf{L}\ b \\
\textsf{get\_sq}\ n\ t\ 0\ i\ j = \textsf{L}\ (\textsf{get}\ n\ t\ i\ j) \\
\textsf{get\_sq}\ (n + 1)\ (\textsf{Q}\ t_0\ t_1\ t_2\ t_3)\ (m + 1)\ i\ j \\
= (\textbf{if}\ i \bmod 2^n + 2^{m+1} \le 2^n \wedge j \bmod 2^n + 2^{m+1} \le 2^n \\
\quad \textbf{then}\ \textsf{get\_sq}\ n\ (\textsf{select}\ (i < 2^n)\ (j < 2^n)\ t_0\ t_1\ t_2\ t_3)\ (m + 1) \\
\qquad\quad (i \bmod 2^n)\ (j \bmod 2^n) \\
\quad \textbf{else}\ \textsf{qf}\ \textsf{Qc}\ (\textsf{get\_sq}\ (n + 1)\ (\textsf{Q}\ t_0\ t_1\ t_2\ t_3)\ m)\ i\ j\ 2^m) \\
\\
\textsf{qf}\ q\ f\ i\ j\ d \equiv q\ (f\ i\ j)\ (f\ i\ (j + d))\ (f\ (i + d)\ j)\ (f\ (i + d)\ (j + d))
\end{array}
$$

Note that in the **else** branch the four subimages do not necessarily come from all four quadrants: the recursive calls are still on the full tree $\textsf{Q}\ t_0\ t_1\ t_2\ t_3$ but reduce the size of the subimage until it fits into a single quadrant (or $\textsf{L}$ is reached).

Although we have explained *get_sq* graphically, it works for any quadtree, not just boolean ones. Functional correctness is expressed like this: pixel (i', j') in the image extracted at (i, j) is the same as pixel $(i + i', j + j')$ in the original image.

$$
\begin{array}{l}
\textsf{height}\ t \le n \wedge i + 2^m \le 2^n \wedge j + 2^m \le 2^n \wedge i' < 2^m \wedge j' < 2^m \longrightarrow \\
\textsf{get}\ m\ (\textsf{get\_sq}\ n\ t\ m\ i\ j)\ i'\ j' = \textsf{get}\ n\ t\ (i + i')\ (j + j')
\end{array}
$$

$$\textsf{height}\ t \le n \wedge \textsf{compressed}\ t \longrightarrow \textsf{compressed}\ (\textsf{get\_sq}\ n\ t\ m\ i\ j)$$

The first correctness theorems requires that the extracted subimage must lie completely within the original image. In contrast, the compression property is simple enough that it does not require this precondition.

13.1.4 From Tree to Matrix and Back

Finally, we may also want to convert between quadtrees and some external format. An obvious candidate is a matrix represented by a list of lists:

$$\textbf{type\_synonym}\ {'a}\ \mathit{mx} = {'a}\ \mathit{list}\ \mathit{list}$$

Function *mx_of* converts a quadtree into a matrix:

$$
\begin{array}{l}
\textsf{mx\_of} :: \mathit{nat} \Rightarrow {'a}\ \mathit{qtree} \Rightarrow {'a}\ \mathit{mx} \\
\textsf{mx\_of}\ n\ (\textsf{L}\ x) = \textsf{replicate}\ 2^n\ (\textsf{replicate}\ 2^n\ x) \\
\textsf{mx\_of}\ (n + 1)\ (\textsf{Q}\ t_0\ t_1\ t_2\ t_3)
\end{array}
$$

$$= \mathit{Qmx}\ (\mathit{mx\_of}\ n\ t_0)\ (\mathit{mx\_of}\ n\ t_1)\ (\mathit{mx\_of}\ n\ t_2)\ (\mathit{mx\_of}\ n\ t_3)$$

$$\mathit{Qmx} :: {'a}\ \mathit{mx} \Rightarrow {'a}\ \mathit{mx} \Rightarrow {'a}\ \mathit{mx} \Rightarrow {'a}\ \mathit{mx} \Rightarrow {'a}\ \mathit{mx}$$

$$\mathit{Qmx}\ mx_0\ mx_1\ mx_2\ mx_3 = \mathit{map2}\ (@)\ mx_0\ mx_1\ @\ \mathit{map2}\ (@)\ mx_2\ mx_3$$

$$\mathit{map2}\ f\ [x_1,\ldots,x_m]\ [y_1,\ldots,y_n] = [f\ x_1\ y_1,\ \ldots,\ f\ x_k\ y_k]\ \text{ where } k = \mathit{min}\ m\ n$$

For example, $\mathit{mx\_of}\ 1\ (Q\ (L\ 0)\ (L\ 1)\ (L\ 2)\ (L\ 3)) = [[0,\ 1],\ [2,\ 3]]$, which we can regard as a two dimensional image:

$$\begin{array}{lll} [& [0,1] & , \\ & [2,3] &] \end{array}$$

This is a 90° rotation of (13.1) and Figure 13.1 where (0,0) is the lower left corner, now it is the upper left one. This is necessary because we want to address a point (i,j) in some mx by $mx\ !\ i\ !\ j$. With the above definition of $\mathit{mx\_of}$ this works. For example, $[[0,\ 1],\ [2,\ 3]]\ !\ 0\ !\ 1 = 1$ and $[[0,\ 1],\ [2,\ 3]]\ !\ 1\ !\ 0 = 2$. In general we can prove that indexing the matrix yields the same value as function get:

$$\mathit{height}\ t \le n \wedge (i,\ j) \in \mathit{sq}\ n \longrightarrow \mathit{mx\_of}\ n\ t\ !\ i\ !\ j = \mathit{get}\ n\ t\ i\ j$$

Conversely, we can also translate a matrix into a quadtree:

$$\mathit{qt\_of} :: \mathit{nat} \Rightarrow {'a}\ \mathit{mx} \Rightarrow {'a}\ \mathit{qtree}$$

$$\begin{array}{l} \mathit{qt\_of}\ (n + 1)\ mx \\ = (\textbf{let}\ (mx_0,\ mx_1,\ mx_2,\ mx_3) = \mathit{decomp}\ n\ mx \\ \quad \textbf{in}\ \mathit{Qc}\ (\mathit{qt\_of}\ n\ mx_0)\ (\mathit{qt\_of}\ n\ mx_1)\ (\mathit{qt\_of}\ n\ mx_2)\ (\mathit{qt\_of}\ n\ mx_3)) \\ \mathit{qt\_of}\ 0\ [[x]] = L\ x \end{array}$$

$$\mathit{decomp} :: \mathit{nat} \Rightarrow {'a}\ \mathit{mx} \Rightarrow {'a}\ \mathit{mx} \times {'a}\ \mathit{mx} \times {'a}\ \mathit{mx} \times {'a}\ \mathit{mx}$$

$$\begin{array}{l} \mathit{decomp}\ n\ mx \\ = (\textbf{let}\ mx_{01} = \mathit{take}\ 2^n\ mx;\ mx_{23} = \mathit{drop}\ 2^n\ mx \\ \quad \textbf{in}\ (\mathit{map}\ (\mathit{take}\ 2^n)\ mx_{01},\ \mathit{map}\ (\mathit{drop}\ 2^n)\ mx_{01},\ \mathit{map}\ (\mathit{take}\ 2^n)\ mx_{23}, \\ \qquad \mathit{map}\ (\mathit{drop}\ 2^n)\ mx_{23})) \end{array}$$

Function $\mathit{qt\_of}$ is correct w.r.t. get and yields a compressed tree:

$$\mathit{sq\_mx}\ n\ mx \wedge (i,\ j) \in \mathit{sq}\ n \longrightarrow \mathit{get}\ n\ (\mathit{qt\_of}\ n\ mx)\ i\ j = mx\ !\ i\ !\ j$$

$$\mathit{sq\_mx}\ n\ mx \longrightarrow \mathit{compressed}\ (\mathit{qt\_of}\ n\ mx)$$

where $\mathit{sq\_mx}\ n\ mx = (|mx| = 2^n \wedge (\forall xs \in \mathit{set}\ mx.\ |xs| = 2^n))$.

The matrix correctness proofs depend on the following auxiliary lemmas:

$$\mathsf{height}\ t \le n \longrightarrow \mathsf{sq\_mx}\ n\ (\mathsf{mx\_of}\ n\ t)$$

$$\mathsf{sq\_mx}\ n\ mx \longrightarrow \mathsf{height}\ (\mathsf{qt\_of}\ n\ mx) \le n$$

$$\begin{array}{l}\mathsf{height}\ (\mathsf{Q}\ t_0\ t_1\ t_2\ t_3) \le n \longrightarrow \\ \mathsf{get}\ n\ (\mathsf{Qc}\ t_0\ t_1\ t_2\ t_3)\ i\ j = \mathsf{get}\ n\ (\mathsf{Q}\ t_0\ t_1\ t_2\ t_3)\ i\ j\end{array}$$

Exercise 13.2. Define a function

$$\mathsf{qt\_of\_fun} :: (nat \Rightarrow nat \Rightarrow {'a}) \Rightarrow nat \Rightarrow {'a}\ qtree$$

that converts a matrix represented as a function into a quadtree of the given resolution and prove its functional correctness

$$(i, j) \in \mathsf{sq}\ n \longrightarrow \mathsf{get}\ n\ (\mathsf{qt\_of\_fun}\ f\ n)\ i\ j = f\ i\ j$$

13.2 Matrix Quadtrees

This section is not about quadtrees *per se* but about their usage. The application is the efficient (because easily parallelizable) implementation of matrix operations. It is well-known that many operations on matrices can be expressed very succinctly on block matrices, which are typically depicted like this:

$$\left[\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right]$$

The correspondence to quadtrees is obvious and we will see how matrix addition and multiplication can be implemented easily on quadtrees.

Our abstract type of (real) matrices is simply a function from indices to real numbers:

$$\textbf{type\_synonym}\ ma = nat \Rightarrow nat \Rightarrow real$$

We have chosen a more abstract model of matrices than the one in Section 13.1.4 because the purpose is to state correctness properties and not to implement algorithms.

Functions are in general infinite objects, matrices are restricted to finite dimensions. We model this by requiring matrices to be 0 outside of their dimensions:

$$\mathsf{sq\_ma}\ n\ a \equiv \forall i\ j.\ 2^n \le i \lor 2^n \le j \longrightarrow a\ i\ j = 0$$

The restriction is required for many nontrivial theorems about matrices, but luckily we get away without requiring it in what follows.

How to convert a quadtree into such a matrix is obvious, except that $\mathsf{L}\ x$ has more than one reasonable interpretation. We interpret $\mathsf{L}\ x$ as the diagonal matrix with x everywhere on the diagonal. Thus the abstraction function ma is defined like this:

$$\mathit{ma} :: \mathit{nat} \Rightarrow \mathit{real\ qtree} \Rightarrow \mathit{ma}$$

$$\mathit{ma}\ n\ (\mathsf{L}\ x) = \boldsymbol{D}\ n\ x$$
$$\mathit{ma}\ (n + 1)\ (\mathsf{Q}\ t_0\ t_1\ t_2\ t_3)$$
$$= \mathit{Qma}\ n\ (\mathit{ma}\ n\ t_0)\ (\mathit{ma}\ n\ t_1)\ (\mathit{ma}\ n\ t_2)\ (\mathit{ma}\ n\ t_3)$$

$$\boldsymbol{D} :: \mathit{nat} \Rightarrow \mathit{real} \Rightarrow \mathit{ma}$$
$$\boldsymbol{D}\ n\ x = \mathit{mk\_sq}\ n\ (\lambda i\ j.\ \textbf{if}\ i = j\ \textbf{then}\ x\ \textbf{else}\ 0)$$

$$\mathit{mk\_sq} :: \mathit{nat} \Rightarrow \mathit{ma} \Rightarrow \mathit{ma}$$
$$\mathit{mk\_sq}\ n\ a = (\lambda i\ j.\ \textbf{if}\ i < 2^n \wedge j < 2^n\ \textbf{then}\ a\ i\ j\ \textbf{else}\ 0)$$

$$\mathit{Qma} :: \mathit{nat} \Rightarrow \mathit{ma} \Rightarrow \mathit{ma} \Rightarrow \mathit{ma} \Rightarrow \mathit{ma} \Rightarrow \mathit{ma}$$
$$\mathit{Qma}\ n\ a\ b\ c\ d$$
$$= (\lambda i\ j.\ \textbf{if}\ i < 2^n\ \textbf{then if}\ j < 2^n\ \textbf{then}\ a\ i\ j\ \textbf{else}\ b\ i\ (j - 2^n)$$
$$\textbf{else if}\ j < 2^n\ \textbf{then}\ c\ (i - 2^n)\ j\ \textbf{else}\ d\ (i - 2^n)\ (j - 2^n))$$

As before, we need to supply the resolution n to obtain a matrix of dimension $2^n \times 2^n$ and to restrict the diagonal matrix $\boldsymbol{D}$ to a square. Note that the correspondence of the four subtrees of Q to the submatrices is not like in (13.1) but like this,

0	1
2	3

assuming the standard notation for matrices, where the upper left corner is the element with index $(0,\ 0)$.

13.2.1 Addition and Multiplication of Matrices

First we define matrix addition and multiplication on abstract functional matrices, then we implement both operations on quadtrees and finally we show the correctness of the implementation via the abstraction function ma.

On the level of matrices, addition and multiplication are defined as in mathematics:

$$(+) :: ma \Rightarrow ma \Rightarrow ma$$
$$a + b = (\lambda i\, j.\ a\ i\ j + b\ i\ j)$$

$$\textit{mult\_ma} :: nat \Rightarrow ma \Rightarrow ma \Rightarrow ma$$
$$a *_n b = (\lambda i\, j.\ \sum k = 0..{<}2^n.\ a\ i\ k \cdot b\ k\ j)$$

Because the dimension of a matrix is implicit, but matrix multiplication depends on it, it is supplied as a subscript in $a *_n b$.

The following lemma collection is easily proved and is used implicitly below:

$$\begin{array}{l}
\boldsymbol{D}\ n\ x + \boldsymbol{D}\ n\ y = \boldsymbol{D}\ n\ (x + y)\\
\boldsymbol{D}\ n\ 0 + a = a\\
a + \boldsymbol{D}\ n\ 0 = a\\
\boldsymbol{D}\ n\ 0 *_n a = \boldsymbol{D}\ n\ 0\\
a *_n \boldsymbol{D}\ n\ 0 = \boldsymbol{D}\ n\ 0\\
\boldsymbol{D}\ n\ x *_n \boldsymbol{D}\ n\ y = \boldsymbol{D}\ n\ (x \cdot y)
\end{array}$$

13.2.2 Addition and Multiplication of Quadtrees

Matrices are represented by quadtrees over real numbers. As before, we have *Qc*, a smart version of *Q* that is used when creating a quadtree. It compresses the four quadrants if they form a diagonal:

$$\begin{array}{l}
\textit{Qc} :: real\ qtree \Rightarrow real\ qtree \Rightarrow real\ qtree \Rightarrow real\ qtree \Rightarrow real\ qtree\\
\textit{Qc}\ (\textit{L}\ x_0)\ (\textit{L}\ x_1)\ (\textit{L}\ x_2)\ (\textit{L}\ x_3)\\
= (\textbf{if}\ x_1 = 0 \wedge x_2 = 0 \wedge x_0 = x_3\ \textbf{then}\ \textit{L}\ x_0\\
\quad \textbf{else}\ \textit{Q}\ (\textit{L}\ x_0)\ (\textit{L}\ x_1)\ (\textit{L}\ x_2)\ (\textit{L}\ x_3))\\
\textit{Qc}\ t_0\ t_1\ t_2\ t_3 = \textit{Q}\ t_0\ t_1\ t_2\ t_3
\end{array}$$

A quadtree is compressed if it does not contain a compressible *Q*:

$$\begin{array}{l}
\textit{compressed} :: real\ qtree \Rightarrow bool\\
\textit{compressed}\ (\textit{L}\ \_) = \textit{True}\\
\textit{compressed}\ (\textit{Q}\ (\textit{L}\ x_0)\ (\textit{L}\ x_1)\ (\textit{L}\ x_2)\ (\textit{L}\ x_3))\\
= (\neg\ (x_1 = 0 \wedge x_2 = 0 \wedge x_0 = x_3))\\
\textit{compressed}\ (\textit{Q}\ t_0\ t_1\ t_2\ t_3)\\
= (\textit{compressed}\ t_0 \wedge \textit{compressed}\ t_1 \wedge \textit{compressed}\ t_2 \wedge \textit{compressed}\ t_3)
\end{array}$$

Addition and multiplication on quadtrees is defined as follows:

$$(\oplus) :: \textit{real qtree} \Rightarrow \textit{real qtree} \Rightarrow \textit{real qtree}$$
$$\mathsf{Q}\ s_0\ s_1\ s_2\ s_3 \oplus \mathsf{Q}\ t_0\ t_1\ t_2\ t_3 = \mathsf{Qc}\ (s_0 \oplus t_0)\ (s_1 \oplus t_1)\ (s_2 \oplus t_2)\ (s_3 \oplus t_3)$$
$$\mathsf{L}\ x \oplus \mathsf{L}\ y = \mathsf{L}\ (x + y)$$
$$\mathsf{L}\ x \oplus \mathsf{Q}\ t_0\ t_1\ t_2\ t_3 = \mathsf{Qc}\ (\mathsf{L}\ x \oplus t_0)\ t_1\ t_2\ (\mathsf{L}\ x \oplus t_3)$$
$$\mathsf{Q}\ t_0\ t_1\ t_2\ t_3 \oplus \mathsf{L}\ x = \mathsf{Qc}\ (t_0 \oplus \mathsf{L}\ x)\ t_1\ t_2\ (t_3 \oplus \mathsf{L}\ x)$$

$$(\otimes) :: \textit{real qtree} \Rightarrow \textit{real qtree} \Rightarrow \textit{real qtree}$$
$$\mathsf{Q}\ s_0\ s_1\ s_2\ s_3 \otimes \mathsf{Q}\ t_0\ t_1\ t_2\ t_3$$
$$= \mathsf{Qc}\ (s_0 \otimes t_0 \oplus s_1 \otimes t_2)\ (s_0 \otimes t_1 \oplus s_1 \otimes t_3)\ (s_2 \otimes t_0 \oplus s_3 \otimes t_2)$$
$$\quad (s_2 \otimes t_1 \oplus s_3 \otimes t_3)$$
$$\mathsf{L}\ x \otimes \mathsf{Q}\ t_0\ t_1\ t_2\ t_3 = \mathsf{Qc}\ (\mathsf{L}\ x \otimes t_0)\ (\mathsf{L}\ x \otimes t_1)\ (\mathsf{L}\ x \otimes t_2)\ (\mathsf{L}\ x \otimes t_3)$$
$$\mathsf{Q}\ t_0\ t_1\ t_2\ t_3 \otimes \mathsf{L}\ x = \mathsf{Qc}\ (t_0 \otimes \mathsf{L}\ x)\ (t_1 \otimes \mathsf{L}\ x)\ (t_2 \otimes \mathsf{L}\ x)\ (t_3 \otimes \mathsf{L}\ x)$$
$$\mathsf{L}\ x \otimes \mathsf{L}\ y = \mathsf{L}\ (x \cdot y)$$

The Q-Q and L-L cases follow the standard definition of how block matrices are added and multiplied. The Q-L and L-Q cases are dealt with by implicitly expanding $\mathsf{L}\ x$ to $\mathsf{Q}\ (\mathsf{L}\ x)\ (\mathsf{L}\ 0)\ (\mathsf{L}\ 0)\ (\mathsf{L}\ x)$ and following the Q-Q case while simplifying addition and multiplication with 0.

Correctness is expressed by showing that the quadtree operations correctly implement the abstract matrix operations via the abstraction function ma:

$$\mathsf{height}\ s \le n \wedge \mathsf{height}\ t \le n \longrightarrow \mathsf{ma}\ n\ (s \oplus t) = \mathsf{ma}\ n\ s + \mathsf{ma}\ n\ t$$

$$\mathsf{height}\ s \le n \wedge \mathsf{height}\ t \le n \longrightarrow \mathsf{ma}\ n\ (s \otimes t) = \mathsf{ma}\ n\ s *_n \mathsf{ma}\ n\ t$$

Moreover, both operations preserve compression:

$$\mathsf{compressed}\ s \wedge \mathsf{compressed}\ t \longrightarrow \mathsf{compressed}\ (s \oplus t)$$

$$\mathsf{compressed}\ s \wedge \mathsf{compressed}\ t \longrightarrow \mathsf{compressed}\ (s \otimes t)$$

The proofs employ the following lemmas:

$$\mathsf{ma}\ (n + 1)\ (\mathsf{Qc}\ t_0\ t_1\ t_2\ t_3) = \mathsf{ma}\ (n + 1)\ (\mathsf{Q}\ t_0\ t_1\ t_2\ t_3)$$

$$\mathsf{Qma}\ n\ a\ b\ c\ d + \mathsf{Qma}\ n\ a'\ b'\ c'\ d'$$
$$= \mathsf{Qma}\ n\ (a + a')\ (b + b')\ (c + c')\ (d + d')$$

$$\boldsymbol{D}\ (n + 1)\ x + \mathsf{Qma}\ n\ a\ b\ c\ d = \mathsf{Qma}\ n\ (\boldsymbol{D}\ n\ x + a)\ b\ c\ (\boldsymbol{D}\ n\ x + d)$$

$$\mathsf{compressed}\ (\mathsf{Qc}\ t_0\ t_1\ t_2\ t_3)$$
$$= (\mathsf{compressed}\ t_0 \wedge \mathsf{compressed}\ t_1 \wedge \mathsf{compressed}\ t_2 \wedge \mathsf{compressed}\ t_3)$$

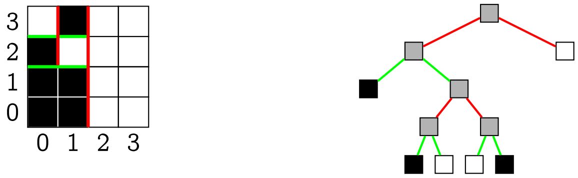


Figure 13.3 Image and corresponding k-d tree

$$\mathit{Qma}\ n\ a\ b\ c\ d\ *_{n\,+\,1}\ \mathit{Qma}\ n\ a'\ b'\ c'\ d'$$
$$= \mathit{Qma}\ n\ (a *_n a' + b *_n c')\ (a *_n b' + b *_n d')\ (c *_n a' + d *_n c')$$
$$\quad (c *_n b' + d *_n d')$$

$$\boldsymbol{D}\ (n + 1)\ x = \mathit{Qma}\ n\ (\boldsymbol{D}\ n\ x)\ (\boldsymbol{D}\ n\ 0)\ (\boldsymbol{D}\ n\ 0)\ (\boldsymbol{D}\ n\ x)$$

$$\mathit{height}\ (\mathit{Qc}\ t_0\ t_1\ t_2\ t_3) \le \mathit{height}\ (Q\ t_0\ t_1\ t_2\ t_3)$$

$$\mathit{height}\ (s \oplus t) \le \mathit{max}\ (\mathit{height}\ s)\ (\mathit{height}\ t)$$

$$\mathit{height}\ (s \otimes t) \le \mathit{max}\ (\mathit{height}\ s)\ (\mathit{height}\ t)$$

13.3 k-Dimensional Region Trees

The direct generalization of quadtrees to k-dimensional space is to subdivide a **hypercube** of resolution $n+1$ into 2^k subcubes of resolution n. We subdivide space with binary splits, dimension by dimension. This means we subdivide a hypercube into two **boxes** (or **hyperrectangles**) along the first dimension, and then subdivide those along the second dimension, and so on, until we reach the last dimension and restart, or a homogeneous box has been obtained. If we start with a hypercube and cycle through all dimension, we end up with another hypercube, but if we stop beforehand, it is some box. An example is shown in Figure 13.3. The first split is always vertical (in red), the second one horizontal (in green). The order or the subtrees is left-right and below-above the split, i.e. in increasing order of coordinates. After the first split, the right rectangle is homogeneous and we do not split it any further.

A **k-d (region) tree** is a binary tree whose leaves are boxes:

```
datatype 'a kdt = Box 'a | Split ('a kdt) ('a kdt)
```

Subtrees of a binary tree can be addressed by a sequence of left-right turns, which we represent as a *bool list*, where *False* represents left.

$$\mathit{subtree} :: {'a}\ \mathit{kdt} \Rightarrow \mathit{bool\ list} \Rightarrow {'a}\ \mathit{kdt}$$

$$\begin{array}{l}
\mathit{subtree}\ t\ [] = t\\
\mathit{subtree}\ (\mathit{Box}\ x)\ \_ = \mathit{Box}\ x\\
\mathit{subtree}\ (\mathit{Split}\ l\ r)\ (b\ \#\ bs) = \mathit{subtree}\ (\textbf{if}\ b\ \textbf{then}\ r\ \textbf{else}\ l)\ bs
\end{array}$$

This is the generalization of function *select* for quadtrees.

13.3.1 Compression

A k-d tree is *compressed* if no two adjacent boxes can be merged:

$$\mathit{compressed} :: {'a}\ \mathit{kdt} \Rightarrow \mathit{bool}$$

$$\begin{array}{l}
\mathit{compressed}\ (\mathit{Box}\ \_) = \mathit{True}\\
\mathit{compressed}\ (\mathit{Split}\ l\ r)\\
= (\mathit{compressed}\ l \wedge \mathit{compressed}\ r \wedge (\nexists b.\ l = \mathit{Box}\ b \wedge r = \mathit{Box}\ b))
\end{array}$$

To keep k-d trees compressed, we introduce the compressing constructor *SplitC*:

$$\mathit{SplitC} :: {'a}\ \mathit{kdt} \Rightarrow {'a}\ \mathit{kdt} \Rightarrow {'a}\ \mathit{kdt}$$

$$\begin{array}{l}
\mathit{SplitC}\ (\mathit{Box}\ b_1)\ (\mathit{Box}\ b_2)\\
= (\textbf{if}\ b_1 = b_2\ \textbf{then}\ \mathit{Box}\ b_1\ \textbf{else}\ \mathit{Split}\ (\mathit{Box}\ b_1)\ (\mathit{Box}\ b_2))\\
\mathit{SplitC}\ l\ r = \mathit{Split}\ l\ r
\end{array}$$

The following useful properties are easily proved:

$$\mathit{compressed}\ l \wedge \mathit{compressed}\ r \longrightarrow \mathit{compressed}\ (\mathit{SplitC}\ l\ r)$$

$$1 \le |bs| \longrightarrow \mathit{subtree}\ (\mathit{SplitC}\ l\ r)\ bs = \mathit{subtree}\ (\mathit{Split}\ l\ r)\ bs$$

13.3.2 Functions *get* and *put*

We generalize the idea of the abstraction function for quadtrees. A k-d tree of resolution n represents a k-dimensional hypercube of side-length 2^n which in turn can be seen as a function from coordinates in k-dimensional space to type $'a$, where we represent a coordinate by a $\mathit{nat\ list}$ (of length k). This function from coordinates to $'a$ is defined recursively over the resolution. A coordinate $[i_1, \ldots, i_k] :: \mathit{nat\ list}$ in a k-dimensional hypercube of resolution $n + 1$ is located in a sub-hypercube of resolution n. The sub-hypercube is identified by the top-bits of the coordinate, i.e. $[i_1 < 2^n, \ldots, i_k < 2^n] :: \mathit{bool\ list}$. On the kdt level it is the subtree addressed by this list. The coordinate of the point in the sub-hypercube is $[i_1 \bmod 2^n, \ldots, i_k \bmod 2^n]$ :: *nat list*. This is the full definition of the abstraction function *get*:

$$\begin{array}{l}
\mathit{get} :: \mathit{nat} \Rightarrow {}'a\ \mathit{kdt} \Rightarrow \mathit{nat\ list} \Rightarrow {}'a \\[4pt]
\mathit{get}\ \_\ \ (\mathit{Box}\ b)\ \_\ = b \\
\mathit{get}\ (n + 1)\ t\ \mathit{ps} \\
= \mathit{get}\ n\ (\mathit{subtree}\ t\ (\mathit{map}\ (\lambda i.\ i < 2^n)\ \mathit{ps}))\ (\mathit{map}\ (\lambda i.\ i \bmod 2^n)\ \mathit{ps})
\end{array}$$

Function *put* updates a single point:

$$\begin{array}{l}
\mathit{put} :: \mathit{nat\ list} \Rightarrow {}'a \Rightarrow \mathit{nat} \Rightarrow {}'a\ \mathit{kdt} \Rightarrow {}'a\ \mathit{kdt} \\[4pt]
\mathit{put}\ \_\ \ a\ 0\ (\mathit{Box}\ \_) = \mathit{Box}\ a \\
\mathit{put}\ \mathit{ps}\ a\ (n + 1)\ t \\
= \mathit{modify}\ (\mathit{put}\ (\mathit{map}\ (\lambda i.\ i \bmod 2^n)\ \mathit{ps})\ a\ n)\ (\mathit{map}\ (\lambda i.\ i < 2^n)\ \mathit{ps})\ t \\[12pt]
\mathit{modify} :: ({}'a\ \mathit{kdt} \Rightarrow {}'a\ \mathit{kdt}) \Rightarrow \mathit{bool\ list} \Rightarrow {}'a\ \mathit{kdt} \Rightarrow {}'a\ \mathit{kdt} \\[4pt]
\mathit{modify}\ f\ []\ t = f\ t \\
\mathit{modify}\ f\ (b\ \#\ \mathit{bs})\ (\mathit{Split}\ l\ r) \\
= (\mathbf{if}\ b\ \mathbf{then}\ \mathit{SplitC}\ l\ (\mathit{modify}\ f\ \mathit{bs}\ r)\ \mathbf{else}\ \mathit{SplitC}\ (\mathit{modify}\ f\ \mathit{bs}\ l)\ r) \\
\mathit{modify}\ f\ (b\ \#\ \mathit{bs})\ (\mathit{Box}\ a) \\
= (\mathbf{let}\ t = \mathit{modify}\ f\ \mathit{bs}\ (\mathit{Box}\ a) \\
\quad \mathbf{in\ if}\ b\ \mathbf{then}\ \mathit{SplitC}\ (\mathit{Box}\ a)\ t\ \mathbf{else}\ \mathit{SplitC}\ t\ (\mathit{Box}\ a))
\end{array}$$

Note that when recombining quadrants on the way back up, *Split* is replaced by *SplitC* to take care of possible compressions.

Just like for quadtrees, there are three correctness properties for *put*:

$$\begin{array}{l}
\mathit{height}\ t \le k \cdot n \wedge \mathit{ps} \in \mathit{cube}\ k\ n \wedge \mathit{ps}' \in \mathit{cube}\ k\ n \longrightarrow \\
\mathit{get}\ n\ (\mathit{put}\ \mathit{ps}\ a\ n\ t)\ \mathit{ps}' = (\mathbf{if}\ \mathit{ps}' = \mathit{ps}\ \mathbf{then}\ a\ \mathbf{else}\ \mathit{get}\ n\ t\ \mathit{ps}')
\end{array}$$

$$\mathit{height}\ t \le n \cdot |\mathit{ps}| \longrightarrow \mathit{height}\ (\mathit{put}\ \mathit{ps}\ a\ n\ t) \le n \cdot |\mathit{ps}|$$

$$\mathit{height}\ t \le |\mathit{ps}| \cdot n \wedge \mathit{compressed}\ t \longrightarrow \mathit{compressed}\ (\mathit{put}\ \mathit{ps}\ a\ n\ t)$$

Additional lemmas are needed because *subtree* and *modify* are recursive:

$$\begin{array}{l}
(\forall t.\ \mathit{height}\ t \le nk \longrightarrow \mathit{height}\ (f\ t) \le nk) \wedge \mathit{height}\ t \le |\mathit{bs}| + nk \longrightarrow \\
\mathit{height}\ (\mathit{modify}\ f\ \mathit{bs}\ t) \le |\mathit{bs}| + nk
\end{array}$$

$$\begin{array}{l}
|\mathit{bs}'| = |\mathit{bs}| \longrightarrow \\
\mathit{subtree}\ (\mathit{modify}\ f\ \mathit{bs}\ t)\ \mathit{bs}' \\
= (\mathbf{if}\ \mathit{bs}' = \mathit{bs}\ \mathbf{then}\ f\ (\mathit{subtree}\ t\ \mathit{bs})\ \mathbf{else}\ \mathit{subtree}\ t\ \mathit{bs}')
\end{array}$$

$$\mathit{compressed}\ t \wedge \mathit{compressed}\ (f\ (\mathit{subtree}\ t\ bs)) \longrightarrow$$
$$\mathit{compressed}\ (\mathit{modify}\ f\ bs\ t)$$

$$\mathit{compressed}\ t \longrightarrow \mathit{compressed}\ (\mathit{subtree}\ t\ bs)$$

For quadtrees, the upper bound on the height was n. Now it is $k \cdot n$ because each step from resolution $n{+}1$ to n can take up to k *Split*s.

13.3.3 Boolean Operations

Boolean combinations of boolean k-d trees are straightforward generalizations of their quadtree relatives and we show only union:

```
union :: bool kdt ⇒ bool kdt ⇒ bool kdt
union (Box b) t = (if b then Box True else t)
union t (Box b) = (if b then Box True else t)
union (Split l₁ r₁) (Split l₂ r₂) = SplitC (union l₁ l₂) (union r₁ r₂)
```

Functional correctness

$$\mathit{max}\ (\mathit{height}\ t_1)\ (\mathit{height}\ t_2) \leq |ps| \cdot n \longrightarrow$$
$$\mathit{get}\ n\ (\mathit{union}\ t_1\ t_2)\ ps = (\mathit{get}\ n\ t_1\ ps \vee \mathit{get}\ n\ t_2\ ps)$$

requires a simple lemma for its proof:

$$\mathit{subtree}\ (\mathit{union}\ t_1\ t_2)\ bs = \mathit{union}\ (\mathit{subtree}\ t_1\ bs)\ (\mathit{subtree}\ t_2\ bs)$$

Moreover, we have the same height and compression properties as for quadtrees:

$$\mathit{height}\ (\mathit{union}\ t_1\ t_2) \leq \mathit{max}\ (\mathit{height}\ t_1)\ (\mathit{height}\ t_2)$$

$$\mathit{compressed}\ t_1 \wedge \mathit{compressed}\ t_2 \longrightarrow \mathit{compressed}\ (\mathit{union}\ t_1\ t_2)$$

Chapter Notes

Samet [1984, 1990] and Aluru [2017] have written surveys of the many variations of quadtrees. Wise [1985, 1986, 1987] has published extensively about the representation of block matrices via quadtrees. We follow Wise's initial [Wise 1987] interpretation of leaves as diagonal matrices.

Quadtrees are obviously a special case. There are also Octrees [Meagher 1982], a version for 3-dimensional space. The generalization to k dimensions is due to Bentley [1975] and Friedman et al. [1977], who invented **k-d trees** for storing sets of k-dimensional points. Rau [2019] has formalized k-d trees. In Section 13.3 we transfer k-d trees to region data.

Part III

Priority Queues

14 Priority Queues

Tobias Nipkow

A **priority queue** of linearly ordered elements is like a multiset where one can insert arbitrary elements and remove minimal elements. Its specification as an ADT is shown in Figure 14.1 where $\mathsf{Min\_mset}\ m \equiv \mathsf{Min}\ (\mathsf{set\_mset}\ m)$ and Min yields the minimal element of a finite and non-empty set of linearly ordered elements.

ADT $\mathit{Priority\_Queue} =$

interface

$\mathit{empty} :: {}'q$

$\mathit{insert} :: {}'a \Rightarrow {}'q \Rightarrow {}'q$

$\mathit{del\_min} :: {}'q \Rightarrow {}'q$

$\mathit{get\_min} :: {}'q \Rightarrow {}'a$

abstraction $\mathit{mset} :: {}'q \Rightarrow {}'a\ \mathit{multiset}$
invariant $\mathit{invar} :: {}'q \Rightarrow \mathit{bool}$

specification

$\mathit{mset}\ \mathit{empty} = \{\!\!\}\}$ $\qquad (\mathit{empty})$

$\mathit{invar}\ \mathit{empty}$ $\qquad (\mathit{empty\text{-}inv})$

$\mathit{invar}\ q \longrightarrow \mathit{mset}\ (\mathit{insert}\ x\ q) = \mathit{mset}\ q + \{\!\!\{x\}\!\!\}$ $\qquad (\mathit{insert})$

$\mathit{invar}\ q \longrightarrow \mathit{invar}\ (\mathit{insert}\ x\ q)$ $\qquad (\mathit{insert\text{-}inv})$

$\mathit{invar}\ q \wedge \mathit{mset}\ q \neq \{\!\!\{\}\!\!\}$
$\longrightarrow \mathit{mset}\ (\mathit{del\_min}\ q) = \mathit{mset}\ q - \{\!\!\{\mathit{get\_min}\ q\}\!\!\}$ $\qquad (\mathit{del\_min})$

$\mathit{invar}\ q \wedge \mathit{mset}\ q \neq \{\!\!\{\}\!\!\} \longrightarrow \mathit{invar}\ (\mathit{del\_min}\ q)$ $\qquad (\mathit{del\_min\text{-}inv})$

$\mathit{invar}\ q \wedge \mathit{mset}\ q \neq \{\!\!\{\}\!\!\} \longrightarrow \mathit{get\_min}\ q = \mathsf{Min\_mset}\ (\mathit{mset}\ q)$ $\qquad (\mathit{get\_min})$

Figure 14.1 ADT $\mathit{Priority\_Queue}$

Mergeable priority queues (see Figure 14.2) provide an additional function *merge* (sometimes: *meld* or *union*) with the obvious functionality.

Our priority queues are simplified. The more general version contains elements that are pairs of some item and its priority.

ADT $\mathit{Priority\_Queue\_Merge} = \mathit{Priority\_Queue} +$

interface

$\mathit{merge} :: {}'q \Rightarrow {}'q \Rightarrow {}'q$

specification

$\mathit{invar}\ q_1 \wedge \mathit{invar}\ q_2 \longrightarrow \mathit{mset}\ (\mathit{merge}\ q_1\ q_2) = \mathit{mset}\ q_1 + \mathit{mset}\ q_2$

$\mathit{invar}\ q_1 \wedge \mathit{invar}\ q_2 \longrightarrow \mathit{invar}\ (\mathit{merge}\ q_1\ q_2)$

Figure 14.2 ADT *Priority_Queue_Merge*

Exercise 14.1. Give a list-based implementation of mergeable priority queues with constant-time *get_min* and *del_min*. Verify the correctness of your implementation w.r.t. *Priority_Queue_Merge*.

14.1 Heaps

A popular implementation technique for priority queues are **heaps**, i.e. trees where the minimal element in each subtree is at the root:

$$\mathit{heap} :: {}'a\ \mathit{tree} \Rightarrow \mathit{bool}$$

$$\mathit{heap}\ \langle\rangle = \mathit{True}$$
$$\mathit{heap}\ \langle l,\ m,\ r\rangle = ((\forall x{\in}\mathit{set\_tree}\ l \cup \mathit{set\_tree}\ r.\ m \le x) \wedge \mathit{heap}\ l \wedge \mathit{heap}\ r)$$

Function *mset_tree* extracts the multiset of elements from a tree:

$$\mathit{mset\_tree} :: {}'a\ \mathit{tree} \Rightarrow {}'a\ \mathit{multiset}$$

$$\mathit{mset\_tree}\ \langle\rangle = \{\!\!\}\!\!\}$$
$$\mathit{mset\_tree}\ \langle l,\ a,\ r\rangle = \{\!\!\{a\}\!\!\} + \mathit{mset\_tree}\ l + \mathit{mset\_tree}\ r$$

When verifying a heap-based implementation of priority queues, the invariant *invar* and the abstraction function *mset* in the ADT *Priority_Queue* are instantiated by *heap* and *mset_tree*. The correctness proofs need to talk about both multisets and (because of the *heap* invariant) sets of elements in a heap. We will only show the relevant multiset properties because the set properties follow easily via the fact $\mathit{set\_mset}\ (\mathit{mset\_tree}\ t) = \mathit{set\_tree}\ t$.

Both *empty* and *get_min* have obvious implementations:

$$\mathit{empty} = \langle\rangle$$

$$\mathit{get\_min}\ \langle\_,\ a,\ \_\rangle = a$$

If a heap-based implementation provides a *merge* function (e.g. skew heaps in Chapter 22), then *insert* and *del_min* can be defined like this:

$$\mathit{insert}\ x\ t = \mathit{merge}\ \langle\langle\rangle,\ x,\ \langle\rangle\rangle\ t$$

$$\begin{aligned}
&\mathit{del\_min}\ \langle\rangle = \langle\rangle\\
&\mathit{del\_min}\ \langle l,\ \_,\ r\rangle = \mathit{merge}\ l\ r
\end{aligned}$$

Note that the following tempting definition of *merge* is functionally correct but leads to very unbalanced heaps:

$$\begin{aligned}
&\mathit{merge}\ \langle\rangle\ t = t\\
&\mathit{merge}\ t\ \langle\rangle = t\\
&\mathit{merge}\ (\langle l_1,\ a_1,\ r_1\rangle =: t_1)\ (\langle l_2,\ a_2,\ r_2\rangle =: t_2)\\
&= (\textbf{if}\ a_1 \le a_2\ \textbf{then}\ \langle l_1,\ a_1,\ \mathit{merge}\ r_1\ t_2\rangle\ \textbf{else}\ \langle l_2,\ a_2,\ \mathit{merge}\ t_1\ r_2\rangle)
\end{aligned}$$

Many of the more advanced implementations of heaps focus on improving this merge function. We will see examples of this in the next chapter on leftist heaps, as well as in the chapters on skew heaps and pairing heaps.

Exercise 14.2. Show functional correctness of the above definition of *merge* (w.r.t. *Priority_Queue_Merge*) and prove functional correctness of the implementations of *insert* and *del_min* (w.r.t. *Priority_Queue*).

Exercise 14.3. Define a function *list* from a heap to a sorted list of its elements and prove $\mathit{mset}\ (\mathit{list}\ t) = \mathit{mset\_tree}\ t$ and $\mathit{heap}\ t \longrightarrow \mathit{sorted}\ (\mathit{list}\ t)$. Also prove that *list* has at most quadratic complexity, i.e. $T_{\mathit{list}}\ t \le |t|_1{}^2$ (possibly with additional constants).

Exercise 14.4. Let xs be a list of linearly ordered elements.

- Prove $\exists t.\ \mathit{inorder}\ t = xs \wedge \mathit{heap}\ t$.
- Prove that this tree t is unique if $\mathit{distinct}\ xs$.
- Define a function *heap_of* that constructs t from xs and prove

  $\mathit{inorder}\ (\mathit{heap\_of}\ xs) = xs$ and $\mathit{heap}\ (\mathit{heap\_of}\ xs)$

Chapter Notes

The idea of the heap goes back to Williams [1964] who also coined the name. In imperative implementations, priority queues frequently also provide an operation *decrease_key*: given some direct reference to an element in the priority queue, decrease its element's priority. This is not completely straightforward in a functional language. Lammich and Nipkow [2019] present an implementation, a Priority Search Tree.

15 Leftist Heaps

Tobias Nipkow

Leftist heaps are heaps in the sense of Section 14.1 and implement mergeable priority queues with efficient (logarithmic) access operations. The key idea is to maintain the invariant that at each node the minimal height of the right child is ≤ that of the left child. We represent leftist heaps as augmented trees that store the minimal height in every node:

type_synonym $'a\ \mathit{lheap} = ('a \times \mathit{nat})\ \mathit{tree}$

$\mathsf{mht} :: 'a\ \mathit{lheap} \Rightarrow \mathit{nat}$
$\mathsf{mht}\ \langle\rangle = 0$
$\mathsf{mht}\ \langle\_, (\_, n), \_\rangle = n$

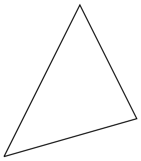
There are two invariants: the standard *heap* invariant (on augmented trees)

$\mathsf{heap} :: ('a \times 'b)\ \mathit{tree} \Rightarrow \mathit{bool}$
$\mathsf{heap}\ \langle\rangle = \mathsf{True}$
$\mathsf{heap}\ \langle l, (m, \_), r\rangle$
$= ((\forall x \in \mathsf{set\_tree}\ l \cup \mathsf{set\_tree}\ r.\ m \le x) \wedge \mathsf{heap}\ l \wedge \mathsf{heap}\ r)$

and the structural invariant that requires that the minimal height of the right child is no bigger than that of the left child (and that the minimal height information in the node is correct):

$\mathsf{ltree} :: 'a\ \mathit{lheap} \Rightarrow \mathit{bool}$
$\mathsf{ltree}\ \langle\rangle = \mathsf{True}$
$\mathsf{ltree}\ \langle l, (\_, n), r\rangle = (\mathsf{mh}\ r \le \mathsf{mh}\ l \wedge n = \mathsf{mh}\ r + 1 \wedge \mathsf{ltree}\ l \wedge \mathsf{ltree}\ r)$

Thus a tree is a **leftist tree** if for every subtree the right spine is a shortest path from the root to a leaf. Pictorially:



Now remember $2^{mh\ t} \leq |t|_1$, i.e. $mh\ t \leq lg\ |t|_1$. Because the expensive operations on leftist heaps descend along the right spine, this means that their running time is logarithmic in the size of the heap.

Exercise 15.1. An alternative definition of leftist tree is via the length of the right spine of the tree:

$$rank :: {'a}\ tree \Rightarrow nat$$

$$rank\ \langle\rangle = 0$$
$$rank\ \langle\_,\ \_,\ r\rangle = rank\ r + 1$$

Prove that the definition by *rank* and by *mh* define the same trees:

$$ltree\_by\ rank\ t = ltree\_by\ mh\ t$$

$$ltree\_by :: ({'a}\ tree \Rightarrow nat) \Rightarrow {'a}\ tree \Rightarrow bool$$

$$ltree\_by\ \_\ \langle\rangle = True$$
$$ltree\_by\ f\ \langle l,\ \_,\ r\rangle = (f\ r \leq f\ l \land ltree\_by\ f\ l \land ltree\_by\ f\ r)$$

It turns out that we can also consider leftist trees by size rather than height and obtain the crucial logarithmic bound for the length of the right spine. Prove

$$ltree\_by\ (\lambda t.\ |t|)\ t \longrightarrow 2^{rank\ t} \leq |t| + 1$$

15.1 Implementation of ADT *Priority_ Queue_ Merge*

The key operation is *merge*:

```
merge :: 'a lheap ⇒ 'a lheap ⇒ 'a lheap
merge ⟨⟩ t = t
merge t ⟨⟩ = t
merge (⟨l₁, (a₁, n₁), r₁⟩ =: t₁) (⟨l₂, (a₂, n₂), r₂⟩ =: t₂)
= (if a₁ ≤ a₂ then node l₁ a₁ (merge r₁ t₂)
   else node l₂ a₂ (merge t₁ r₂))
```

```
node :: 'a lheap ⇒ 'a ⇒ 'a lheap ⇒ 'a lheap
node l a r
= (let mhl = mht l; mhr = mht r
   in if mhr ≤ mhl then ⟨l, (a, mhr + 1), r⟩
      else ⟨r, (a, mhl + 1), l⟩)
```

Termination of *merge* can be proved either by the sum of the sizes of the two arguments (which goes down with every call) or by the lexicographic product of the two size measures: either the first argument becomes smaller or it stays unchanged and the second argument becomes smaller.

As shown in Section 14.1, once we have *merge*, the other operations are easily definable. We repeat the definitions of those operations that change because this chapter employs augmented rather than ordinary trees:

```
get_min :: 'a lheap ⇒ 'a
get_min ⟨_, (a, _), _⟩ = a

insert :: 'a ⇒ 'a lheap ⇒ 'a lheap
insert x t = merge ⟨⟨⟩, (x, 1), ⟨⟩⟩ t
```

15.2 Correctness

The above implementation is proved correct with respect to the ADT *Priority_Queue_Merge* where

```
mset_tree :: ('a × 'b) tree ⇒ 'a multiset
mset_tree ⟨⟩ = {#}
mset_tree ⟨l, (a, _), r⟩ = {#a#} + mset_tree l + mset_tree r

invar t = (heap t ∧ ltree t)
```

Correctness of *get_min* follows directly from the heap invariant:

$$\mathit{heap}\ t \wedge t \neq \langle\rangle \longrightarrow \mathit{get\_min}\ t = \mathit{Min}\ (\mathit{set\_tree}\ t)$$

From the following inductive lemmas about *merge*

$$\mathit{mset\_tree}\ (\mathit{merge}\ t_1\ t_2) = \mathit{mset\_tree}\ t_1 + \mathit{mset\_tree}\ t_2$$

$$\mathit{ltree}\ l \wedge \mathit{ltree}\ r \longrightarrow \mathit{ltree}\ (\mathit{merge}\ l\ r)$$

$$\mathit{heap}\ l \wedge \mathit{heap}\ r \longrightarrow \mathit{heap}\ (\mathit{merge}\ l\ r)$$

correctness of *insert* and *del_min* follow easily:

$$\mathit{mset\_tree}\ (\mathit{insert}\ x\ t) = \mathit{mset\_tree}\ t + \{\!\{x\}\!\}$$

$$\mathit{mset\_tree}\ (\mathit{del\_min}\ t) = \mathit{mset\_tree}\ t - \{\!\{\mathit{get\_min}\ t\}\!\}$$

$$\mathit{ltree}\ t \longrightarrow \mathit{ltree}\ (\mathit{insert}\ x\ t)$$

$$\mathit{heap}\ t \longrightarrow \mathit{heap}\ (\mathit{insert}\ x\ t)$$

$$\mathit{ltree}\ t \longrightarrow \mathit{ltree}\ (\mathit{del\_min}\ t)$$

$$\mathit{heap}\ t \longrightarrow \mathit{heap}\ (\mathit{del\_min}\ t)$$

Of course the above proof (ignoring the *ltree* part) works for any mergeable priority queue implemented as a heap.

15.3 Running Time

The running time functions are shown in Appendix B.5. By induction on the computation of *merge* we obtain

$$\mathit{ltree}\ l \wedge \mathit{ltree}\ r \longrightarrow T_{\mathit{merge}}\ l\ r \le \mathit{mh}\ l + \mathit{mh}\ r + 1$$

With $2^{\mathit{mh}\ t} \le |t|_1$ it follows that

$$\mathit{ltree}\ l \wedge \mathit{ltree}\ r \longrightarrow T_{\mathit{merge}}\ l\ r \le \mathit{lg}\ |l|_1 + \mathit{lg}\ |r|_1 + 1 \tag{15.1}$$

which implies logarithmic bounds for insertion and deletion:

$$\mathit{ltree}\ t \longrightarrow T_{\mathit{insert}}\ x\ t \le \mathit{lg}\ |t|_1 + 2$$

$$\mathit{ltree}\ t \longrightarrow T_{\mathit{del\_min}}\ t \le 2 \cdot \mathit{lg}\ |t|_1 + 1$$

The derivation of the bound for insertion is trivial. The proof of the deletion bound is a simple case analysis (on t).

15.4 Converting a List into a Leftist Heap

We follow the pattern of bottom-up merge sort (Section 2.5) and of the conversions from lists to 2-3 trees (Section 7.3). In both cases we repeatedly pass over a list of objects, merging pairs of adjacent objects in each pass. However, the complexity differs: in merge sort, each merge takes linear time, which leads to the overall complexity of $O(n \lg n)$; when converting a list into a 2-3 tree, each combination of two trees takes only constant time, which leads to a linear overall complexity. So what happens if the merge step takes logarithmic time, as in (15.1)? But first the algorithm, which is very similar to merge sort:

$$\mathit{merge\_adj} :: {}'a\ \mathit{lheap}\ \mathit{list} \Rightarrow {}'a\ \mathit{lheap}\ \mathit{list}$$

$$\begin{aligned}
&\mathit{merge\_adj}\ [] = []\\
&\mathit{merge\_adj}\ [t] = [t]\\
&\mathit{merge\_adj}\ (t_1\ \#\ t_2\ \#\ ts) = \mathit{merge}\ t_1\ t_2\ \#\ \mathit{merge\_adj}\ ts
\end{aligned}$$

$$\mathit{merge\_all} :: {}'a\ \mathit{lheap}\ \mathit{list} \Rightarrow {}'a\ \mathit{lheap}$$

$$\begin{aligned}
&\mathit{merge\_all}\ [] = \langle\rangle\\
&\mathit{merge\_all}\ [t] = t\\
&\mathit{merge\_all}\ ts = \mathit{merge\_all}\ (\mathit{merge\_adj}\ ts)
\end{aligned}$$

$$\mathit{lheap\_list} :: {}'a\ \mathit{list} \Rightarrow {}'a\ \mathit{lheap}$$

$$\mathit{lheap\_list}\ xs = \mathit{merge\_all}\ (\mathit{map}\ (\lambda x.\ \langle\langle\rangle,\ (x,\ 1),\ \langle\rangle\rangle)\ xs)$$

Termination of *merge_all* follows because *merge_adj* decreases the length of the list if $|ts| \geq 2$:

$$|\mathit{merge\_adj}\ ts| = (|ts| + 1)\ \mathrm{div}\ 2$$

Functional correctness is straightforward: from the inductive properties

$$\begin{aligned}
&(\forall t \in \mathit{set}\ ts.\ \mathit{heap}\ t) \longrightarrow (\forall t \in \mathit{set}\ (\mathit{merge\_adj}\ ts).\ \mathit{heap}\ t)\\
&(\forall t \in \mathit{set}\ ts.\ \mathit{heap}\ t) \longrightarrow \mathit{heap}\ (\mathit{merge\_all}\ ts)
\end{aligned}$$

$$\begin{aligned}
&(\forall t \in \mathit{set}\ ts.\ \mathit{ltree}\ t) \longrightarrow (\forall t \in \mathit{set}\ (\mathit{merge\_adj}\ ts).\ \mathit{ltree}\ t)\\
&(\forall t \in \mathit{set}\ ts.\ \mathit{ltree}\ t) \longrightarrow \mathit{ltree}\ (\mathit{merge\_all}\ ts)
\end{aligned}$$

$$\begin{aligned}
&\textstyle\sum_{\#}\ (\mathit{image\_mset}\ \mathit{mset\_tree}\ (\mathit{mset}\ (\mathit{merge\_adj}\ ts)))\\
&= \textstyle\sum_{\#}\ (\mathit{image\_mset}\ \mathit{mset\_tree}\ (\mathit{mset}\ ts))\\
&\mathit{mset\_tree}\ (\mathit{merge\_all}\ ts) = \textstyle\sum_{\#}\ (\mathit{mset}\ (\mathit{map}\ \mathit{mset\_tree}\ ts))
\end{aligned}$$

it follows directly that *lheap_list* xs yields a leftist heap with the same multiset of elements as in xs:

$$\mathit{heap}\ (\mathit{lheap\_list}\ ts)$$

$$\mathit{ltree}\ (\mathit{lheap\_list}\ ts)$$

$$\mathit{mset\_tree}\ (\mathit{lheap\_list}\ xs) = \mathit{mset}\ xs$$

The running time analysis is more interesting. We only count the time for *merge* to keep things simple.

$$
\begin{array}{l}
T_{\mathit{merge\_adj}} :: \ \mathit{'a\ lheap\ list} \Rightarrow \mathit{nat} \\
T_{\mathit{merge\_adj}}\ [] = 0 \\
T_{\mathit{merge\_adj}}\ [\_] = 0 \\
T_{\mathit{merge\_adj}}\ (t_1\ \#\ t_2\ \#\ ts) = T_{\mathit{merge}}\ t_1\ t_2 + T_{\mathit{merge\_adj}}\ ts
\end{array}
$$

The remaining time functions are displayed in Appendix B.5.

To simplify things further we assume that the length of the initial list xs and thus the length of all intermediate lists of heaps are powers of 2 and in any of the intermediate lists all heaps have the same size.

Because the complexity of *merge* is logarithmic in the size of the two heaps (15.1), the following upper bound for *merge_adj* follows by an easy computation induction:

$$
\begin{array}{l}
(\forall t \in \mathit{set}\ ts.\ \mathit{ltree}\ t) \wedge (\forall t \in \mathit{set}\ ts.\ |t| = n) \longrightarrow \\
T_{\mathit{merge\_adj}}\ ts \le (|ts| \operatorname{div} 2) \cdot \mathit{Tm}\ n
\end{array}
$$

where $\mathit{Tm}\ n \equiv 2 \cdot \mathit{lg}\ (n + 1) + 1$.

The complexity of *merge_all* can be expressed as a sum:

$$
\begin{array}{l}
(\forall t \in \mathit{set}\ ts.\ \mathit{ltree}\ t) \wedge (\forall t \in \mathit{set}\ ts.\ |t| = n) \wedge |ts| = 2^k \longrightarrow \\
T_{\mathit{merge\_all}}\ ts \le (\sum_{i = 1}^{k} 2^{k - i} \cdot \mathit{Tm}\ (2^{i - 1} \cdot n))
\end{array}
\tag{15.2}
$$

Each summand is the complexity of one *merge_adj* call on heap lists whose lengths go down from 2^k to 2 and whose heaps go up in size from n to $2^{k - 1} \cdot n$. The proof is by induction on the computation of *merge_all*.

The following lemma will permit us to find a closed upper bound for the sum in (15.2). The proof is a straightforward induction on k.

Lemma 15.1. $(\sum_{i = 1}^{k} 2^{k - i} \cdot (2 \cdot i + 1)) = 5 \cdot 2^k - 2 \cdot k - 5$

Now we can upper-bound $T_{\mathit{lheap\_list}}$ as follows if $|xs| = 2^k$:

$$
\begin{array}{ll}
T_{\mathit{lheap\_list}}\ xs = T_{\mathit{merge\_all}}\ (\mathit{map}\ (\lambda x.\ \langle\langle\rangle, (x, 1), \langle\rangle\rangle)\ xs) & \\
\le \sum_{i = 1}^{k} 2^{k - i} \cdot \mathit{Tm}\ (2^{i - 1}) & \text{by (15.2) (where } n = 1\text{) and } |xs| = 2^k \\
\le \sum_{i = 1}^{k} 2^{k - i} \cdot (2 \cdot \mathit{lg}\ (2 \cdot 2^{i - 1}) + 1) & \\
= \sum_{i = 1}^{k} 2^{k - i} \cdot (2 \cdot i + 1) & \\
= 5 \cdot 2^k - 2 \cdot k - 5 & \text{by Lemma 15.1}
\end{array}
$$

Thus (15.2) implies that $T_{\mathit{lheap\_list}}\ xs$ is upper-bounded by a function linear in $|xs|$:

$$
|xs| = 2^k \longrightarrow T_{\mathit{lheap\_list}}\ xs \le 5 \cdot |xs| - 2 \cdot \mathit{lg}\ |xs|
$$

The assumption $|xs| = 2^k$ merely simplifies technicalities. With more care one can show that $T_{\mathit{lheap\_list}} \in O(n)$ holds for all inputs of length n; the term $- 2 \cdot \mathit{lg}\ |xs|$ is irrelevant because $O(n - \lg n) = O(n)$.

Finally note that the above complexity analysis has nothing to do with leftist heaps or priority queues and works for any *merge* function of the given logarithmic complexity. Our proofs generalize easily. One can even go one step further and show that *merge_all* has linear complexity as long as *merge* has sublinear complexity. This is a special case of the **master theorem** [Cormen et al. 2009] for divide-and-conquer algorithms, because *merge_all* is just divide-and-conquer in reverse. However, proving even this special case (let alone the full master theorem) is much harder than the proofs above.

Exercise 15.2. Define a tail-recursive variant of *merge_adj*

$$\textit{merge\_adj2} :: {'a}\ \textit{lheap list} \Rightarrow {'a}\ \textit{lheap list} \Rightarrow {'a}\ \textit{lheap list}$$

(with the same complexity as *merge_adj*, in particular no (@)) and define new variants *merge_all2* and *lheap_list2* of *merge_all* and *lheap_list* that utilize *merge_adj2*. Prove functional correctness of *lheap_list2*:

$$\textit{mset\_tree}\ (\textit{lheap\_list2}\ xs) = \textit{mset}\ xs$$
$$\textit{heap}\ (\textit{lheap\_list2}\ ts) \qquad \textit{ltree}\ (\textit{lheap\_list2}\ ts)$$

Note that $\textit{merge\_adj2}\ []\ ts = \textit{merge\_adj}\ ts$ is not required.

Chapter Notes

Leftist heaps were invented by Crane [1972]. Another version of leftist trees, based on weight rather than height, was introduced by Cho and Sahni [1998].

16 Priority Queues via Braun Trees

Tobias Nipkow

In Chapter 11 we introduced Braun trees and showed how to implement arrays. In the current chapter we show how to implement priority queues by means of Braun trees. Because Braun trees have logarithmic height this guarantees logarithmic running times for insertion and deletion. Remember that every node $\langle l, x, r\rangle$ in a Braun tree satisfies $|l| = |r| \vee |l| = |r| + 1$ $(*)$.

16.1 Implementation of ADT *Priority_Queue*

We follow the heap approach in Section 14.1. Functions *empty*, *get_min*, *heap* and *mset_tree* are defined as in that section.

Insertion and deletion maintain the Braun tree property $(*)$ by inserting into the right (and possibly smaller) child, deleting from the left (and possibly larger) child, and swapping children to reestablish $(*)$.

Insertion is easy and clearly maintains both the heap and the Braun tree property:

```
insert :: 'a ⇒ 'a tree ⇒ 'a tree
insert a ⟨⟩ = ⟨⟨⟩, a, ⟨⟩⟩
insert a ⟨l, x, r⟩
= (if a < x then ⟨insert x r, a, l⟩ else ⟨insert a r, x, l⟩)
```

To delete the minimal (i.e. root) element from a tree, extract the leftmost element from the tree and let it sift down to its correct position in the tree *à la* heapsort:

```
del_min :: 'a tree ⇒ 'a tree
del_min ⟨⟩ = ⟨⟩
del_min ⟨⟨⟩, _, _⟩ = ⟨⟩
del_min ⟨l, _, r⟩ = (let (y, l') = del_left l in sift_down r y l')
```

```
del_left :: 'a tree ⇒ 'a × 'a tree
del_left ⟨⟨⟩, x, r⟩ = (x, r)
del_left ⟨l, x, r⟩ = (let (y, l') = del_left l in (y, ⟨r, x, l'⟩))

sift_down :: 'a tree ⇒ 'a ⇒ 'a tree ⇒ 'a tree
sift_down ⟨⟩ a _ = ⟨⟨⟩, a, ⟨⟩⟩
sift_down ⟨⟨⟩, x, _⟩ a ⟨⟩
= (if a ≤ x then ⟨⟨⟨⟩, x, ⟨⟩⟩, a, ⟨⟩⟩ else ⟨⟨⟨⟩, a, ⟨⟩⟩, x, ⟨⟩⟩)
sift_down (⟨l₁, x₁, r₁⟩ =: t₁) a (⟨l₂, x₂, r₂⟩ =: t₂)
= (if a ≤ x₁ ∧ a ≤ x₂ then ⟨t₁, a, t₂⟩
   else if x₁ ≤ x₂ then ⟨sift_down l₁ a r₁, x₁, t₂⟩
        else ⟨t₁, x₂, sift_down l₂ a r₂⟩)
```

In the first two equations for *sift_down*, the Braun tree property guarantees that the "_" arguments must be empty trees if the pattern matches.

Termination of *sift_down* can be proved with the help of a measure function depending on the two tree arguments l and r. A simple measure that works is $|l| + |r|$ but it is overly pessimistic. A better measure is $\mathit{max}\ (h\ l)\ (h\ r)$ because it is a tight upper bound on the number of steps to termination. Thus it yields a better upper bound for the later running time analysis.

16.2 Correctness

We outline the correctness proofs for *insert* and *del_min* by presenting the key lemmas. Correctness of *insert* is straightforward:

$$|\mathit{insert}\ x\ t| = |t| + 1$$

$$\mathit{mset\_tree}\ (\mathit{insert}\ x\ t) = \{\!\{x\}\!\} + \mathit{mset\_tree}\ t$$

$$\mathit{braun}\ t \longrightarrow \mathit{braun}\ (\mathit{insert}\ x\ t)$$

$$\mathit{heap}\ t \longrightarrow \mathit{heap}\ (\mathit{insert}\ x\ t)$$

Correctness of *del_min* builds on analogous correctness lemmas for the auxiliary functions:

$$\mathit{del\_left}\ t = (x,\ t') \land t \neq \langle\rangle \longrightarrow \mathit{mset\_tree}\ t = \{\!\{x\}\!\} + \mathit{mset\_tree}\ t'$$

$$\mathit{del\_left}\ t = (x,\ t') \land t \neq \langle\rangle \land \mathit{heap}\ t \longrightarrow \mathit{heap}\ t'$$

$$\mathit{del\_left}\ t = (x,\ t') \land t \neq \langle\rangle \longrightarrow |t| = |t'| + 1 \tag{16.1}$$

$$\mathit{del\_left}\ t = (x,\ t') \land t \neq \langle\rangle \land \mathit{braun}\ t \longrightarrow \mathit{braun}\ t' \tag{16.2}$$

$$\mathit{braun}\ \langle l,\ a,\ r\rangle \longrightarrow |\mathit{sift\_down}\ l\ a\ r| = |l| + |r| + 1$$

$$\mathit{braun}\ \langle l,\ a,\ r\rangle \longrightarrow \mathit{braun}\ (\mathit{sift\_down}\ l\ a\ r)$$

$$\mathit{braun}\ \langle l,\ a,\ r\rangle \longrightarrow$$
$$\mathit{mset\_tree}\ (\mathit{sift\_down}\ l\ a\ r) = \{\!\{a\}\!\} + (\mathit{mset\_tree}\ l + \mathit{mset\_tree}\ r)$$

$$\mathit{braun}\ \langle l,\ a,\ r\rangle \wedge \mathit{heap}\ l \wedge \mathit{heap}\ r \longrightarrow \mathit{heap}\ (\mathit{sift\_down}\ l\ a\ r)$$

$$\mathit{braun}\ t \longrightarrow \mathit{braun}\ (\mathit{del\_min}\ t)$$

$$\mathit{heap}\ t \wedge \mathit{braun}\ t \longrightarrow \mathit{heap}\ (\mathit{del\_min}\ t)$$

$$\mathit{braun}\ t \wedge t \neq \langle\rangle \longrightarrow \mathit{mset\_tree}\ (\mathit{del\_min}\ t) = \mathit{mset\_tree}\ t - \{\!\{\mathit{get\_min}\ t\}\!\}$$

16.3 Running Time

The running time functions are shown in Appendix B.6. Intuitively, all operations are linear in the height of the tree, which in turn is logarithmic in the number of elements (see Section 11.2).

Upper bounds for the running times of *insert*, *del_left* and *sift_down* are proved by straightforward inductions:

$$T_{\mathit{insert}}\ a\ t \leq h\ t + 1$$

$$t \neq \langle\rangle \longrightarrow T_{\mathit{del\_left}}\ t \leq h\ t \tag{16.3}$$

$$\mathit{braun}\ \langle l,\ a,\ r\rangle \longrightarrow T_{\mathit{sift\_down}}\ l\ x\ r \leq \mathit{max}\ (h\ l)\ (h\ r) + 1 \tag{16.4}$$

The analysis of *del_min* requires a bit more work, including another auxiliary inductive fact:

$$\mathit{del\_left}\ t = (x,\ t') \wedge t \neq \langle\rangle \longrightarrow h\ t' \leq h\ t \tag{16.5}$$

Lemma 16.1. $\mathit{braun}\ t \longrightarrow T_{\mathit{del\_min}}\ t \leq 2 \cdot h\ t$

Proof by induction on t. The base case is trivial. If $t = \langle l,\ x,\ r\rangle$, the case $l = \langle\rangle$ is again trivial. Assume $l \neq \langle\rangle$. The call of *del_min* must yield a pair: $\mathit{del\_left}\ l = (y,\ l')$. Now we are ready for the main derivation:

$$T_{\mathit{del\_min}}\ t = T_{\mathit{del\_left}}\ l + T_{\mathit{sift\_down}}\ r\ y\ l'$$
$$\leq \mathit{height}\ l + T_{\mathit{sift\_down}}\ r\ y\ l' \qquad \text{by (16.3)}$$

In order to upper-bound $T_{\mathit{sift\_down}}\ r\ y\ l'$ via (16.4), we need $\mathit{braun}\ \langle r,\ y,\ l'\rangle$, which follows from $\mathit{braun}\ t$ via (16.2) and (16.1). Thus

$$\leq h\ l + \mathit{max}\ (h\ r)\ (h\ l') + 1$$
$$\leq h\ l + \mathit{max}\ (h\ r)\ (h\ l) + 1 \qquad \text{by (16.5)}$$
$$\leq 2 \cdot \mathit{max}\ (h\ l)\ (h\ r) + 1 \leq 2 \cdot h\ t + 1$$

□

Chapter Notes

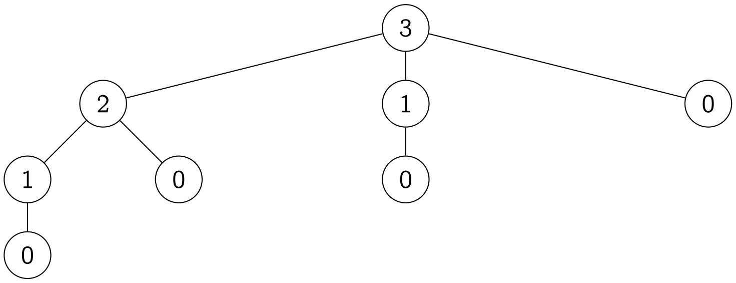
Our implementation of priority queues via Braun trees is due to Paulson [1996] who credits it to Okasaki.

17 Binomial Priority Queues

Peter Lammich

Binomial priority queues are another common implementation of mergeable priority queues that supports efficient ($O(\lg n)$) *insert*, *get_min*, *del_min*, and *merge* operations.

The basic building blocks of a binomial priority queue are **binomial trees**, which are defined recursively as follows: a binomial tree of rank r is a node with r children that are binomial trees of ranks $r - 1, \ldots, 0$, in that order. This is an example of a binomial tree of rank 3:



It can be shown that a binomial tree of rank r has $\binom{r}{l}$ nodes on level l (see Exercise 17.1). Hence the name.

To define binomial trees, we first define a more general datatype and the usual syntax for nodes:

```
datatype 'a tree = Node nat 'a ('a tree list)
⟨r, x, ts⟩ ≡ Node r x ts
```

Apart from the list of children, a node stores a rank and a root element:

```
rank ⟨r, x, ts⟩ = r        root ⟨r, x, ts⟩ = x
```

This datatype contains all binomial trees, but also some non-binomial trees. To carve out the binomial trees, we define an invariant, which reflects the informal definition above:

$$\mathit{btree} :: {'a}\ \mathit{tree} \Rightarrow \mathit{bool}$$
$$\mathit{btree}\ \langle r,\ \_,\ ts\rangle = ((\forall t{\in}\mathit{set}\ ts.\ \mathit{btree}\ t) \wedge \mathit{map}\ \mathit{rank}\ ts = \mathit{rev}\ [0..{<}r])$$

Additionally, we require the heap property, i.e. that the root element of each subtree is a minimal element in that subtree:

$$\mathit{heap} :: {'a}\ \mathit{tree} \Rightarrow \mathit{bool}$$
$$\mathit{heap}\ \langle \_,\ x,\ ts\rangle = (\forall t{\in}\mathit{set}\ ts.\ \mathit{heap}\ t \wedge x \le \mathit{root}\ t)$$

Thus, a **binomial heap** is a tree that satisfies both the structural and the heap invariant. The two invariants are combined into a single predicate:

$$\mathit{bheap} :: {'a}\ \mathit{tree} \Rightarrow \mathit{bool}$$
$$\mathit{bheap}\ t = (\mathit{btree}\ t \wedge \mathit{heap}\ t)$$

A **binomial priority queue** or **binomial forest** is a list of binomial trees

```
type_synonym 'a forest = 'a tree list
```

with strictly ascending rank:

$$\mathit{invar} :: {'a}\ \mathit{forest} \Rightarrow \mathit{bool}$$
$$\mathit{invar}\ ts = ((\forall t{\in}\mathit{set}\ ts.\ \mathit{bheap}\ t) \wedge \mathit{sorted\_wrt}\ ({<})\ (\mathit{map}\ \mathit{rank}\ ts))$$

Note that *sorted_wrt* states that a list is sorted w.r.t. the specified relation, here $(<)$. It is defined in Appendix A.

17.1 Size

The following functions return the multiset of elements in a binomial tree and forest:

$$\mathit{mset\_tree} :: {'a}\ \mathit{tree} \Rightarrow {'a}\ \mathit{multiset}$$
$$\mathit{mset\_tree}\ \langle \_,\ a,\ ts\rangle = \{\!\{a\}\!\} + (\textstyle\sum_{t\in_{\#}\mathit{mset}\ ts} \mathit{mset\_tree}\ t)$$
$$\mathit{mset\_forest} :: {'a}\ \mathit{forest} \Rightarrow {'a}\ \mathit{multiset}$$
$$\mathit{mset\_forest}\ ts = (\textstyle\sum_{t\in_{\#}\mathit{mset}\ ts} \mathit{mset\_tree}\ t)$$

Most operations on binomial forests are linear in the length of the forest. To show that the length is bounded by the logarithm of the number of elements, we first observe that the number of elements in a binomial tree is already determined by its rank. A binomial tree of rank r has 2^r nodes:

$$\textit{btree}\ t \longrightarrow |\textit{mset\_tree}\ t| = 2^{\textit{rank}\ t}$$

This proposition is proved by induction on the tree structure. A tree of rank 0 has one element, and a tree of rank $r{+}1$ has subtrees of rank $0, 1, \ldots, r$. By the induction hypothesis, these have $2^0, 2^1, \ldots, 2^r$ elements, i.e., $2^{r+1} - 1$ elements together. Including the element at the root, there are 2^{r+1} elements.

The length of a binomial forest is bounded logarithmically in the number of its elements:

$$\textit{invar}\ ts \longrightarrow |ts| \leq \textit{lg}\ (|\textit{mset\_forest}\ ts| + 1) \tag{17.1}$$

To prove this, recall that the forest ts is strictly sorted by rank. Thus, we can underestimate the ranks of the trees in ts by $0, 1, \ldots, |ts| - 1$. This means that they must have at least $2^0, 2^1, \ldots, 2^{|ts|-1}$ elements, i.e., at least $2^{|ts|} - 1$ elements together, which yields the desired bound.

17.2 Implementation of ADT *Priority_ Queue*

Obviously, the *empty* binomial forest is [] and a binomial forest *is_empty* iff it is []. Correctness is trivial. The remaining operations are more interesting.

17.2.1 Insertion

A crucial property of binomial trees is that we can link two binomial trees of rank r to form a binomial tree of rank $r + 1$, simply by prepending one tree as the first child of the other. To preserve the heap property, we add the tree with the bigger root element below the tree with the smaller root element. This **linking** of trees is illustrated in Figure 17.1. Formally:

```
link :: 'a tree ⇒ 'a tree ⇒ 'a tree
link (⟨r, x₁, ts₁⟩ =: t₁) (⟨r', x₂, ts₂⟩ =: t₂)
= (if x₁ ≤ x₂ then ⟨r + 1, x₁, t₂ # ts₁⟩ else ⟨r + 1, x₂, t₁ # ts₂⟩)
```

By case distinction, we can easily prove that *link* preserves the invariant and that the resulting tree contains the elements of both arguments.

$$\textit{bheap}\ t_1 \wedge \textit{bheap}\ t_2 \wedge \textit{rank}\ t_1 = \textit{rank}\ t_2 \longrightarrow \textit{bheap}\ (\textit{link}\ t_1\ t_2)$$

$$\textit{mset\_tree}\ (\textit{link}\ t_1\ t_2) = \textit{mset\_tree}\ t_1 + \textit{mset\_tree}\ t_2$$

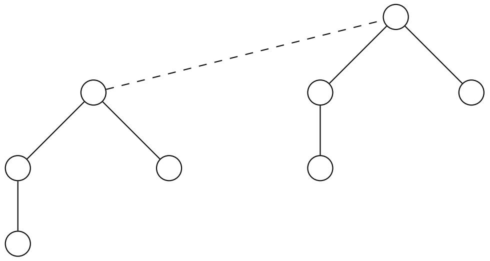


Figure 17.1 Linking two binomial trees of rank 2 to form a binomial tree of rank 3, by linking the left tree as first child of the right tree, as indicated by the dashed line. We assume that the root element of the left tree is greater than or equal to the root element of the right tree, such that the heap property is preserved.

The link operation forms the basis of inserting a tree into a forest: if the forest does not contain a tree with the same rank, we can simply insert the tree at the correct position in the forest. Otherwise, we merge the two trees and recursively insert the result. For our purposes, we can additionally assume that the rank of the tree to be inserted is smaller than or equal to the lowest rank in the forest, which saves us a case in the following definition:

$\mathit{ins\_tree} :: {}'a\ \mathit{tree} \Rightarrow {}'a\ \mathit{forest} \Rightarrow {}'a\ \mathit{forest}$

$\mathit{ins\_tree}\ t\ [] = [t]$
$\mathit{ins\_tree}\ t_1\ (t_2\ \#\ ts)$
$= (\textbf{if}\ \mathit{rank}\ t_1 < \mathit{rank}\ t_2\ \textbf{then}\ t_1\ \#\ t_2\ \#\ ts\ \textbf{else}\ \mathit{ins\_tree}\ (\mathit{link}\ t_1\ t_2)\ ts)$

Invariant preservation and functional correctness of *ins_tree* is easily proved by induction using the respective properties for *link*:

$$\mathit{bheap}\ t \wedge \mathit{invar}\ ts \wedge (\forall t' \in \mathit{set}\ ts.\ \mathit{rank}\ t \le \mathit{rank}\ t') \longrightarrow \mathit{invar}\ (\mathit{ins\_tree}\ t\ ts)$$
$$\mathit{mset\_forest}\ (\mathit{ins\_tree}\ t\ ts) = \mathit{mset\_tree}\ t + \mathit{mset\_forest}\ ts$$

A single element is inserted as a one-element (rank 0) tree:

$\mathit{insert} :: {}'a \Rightarrow {}'a\ \mathit{forest} \Rightarrow {}'a\ \mathit{forest}$

$\mathit{insert}\ x\ ts = \mathit{ins\_tree}\ \langle 0,\ x,\ []\rangle\ ts$

The above definition meets the specification for insert required by the *Priority_ Queue* ADT:

$$\textit{invar}\ t \longrightarrow \textit{invar}\ (\textit{insert}\ x\ t)$$

$$\textit{mset\_forest}\ (\textit{insert}\ x\ t) = \{\!\!\{x\}\!\!\} + \textit{mset\_forest}\ t$$

17.2.2 Merging

Recall the merge algorithm used in top-down merge sort (Section 2.4). It merges two sorted lists by repeatedly taking the smaller list head. We proceed analogously when merging forests, where "smaller" means "of smaller rank". If both ranks are equal, we link the two heads (call the result t') and insert t' into the result ts' of the recursive call of merge. Thus, the resulting forest will be strictly ordered by rank. Formally:

$$\textit{merge} :: \textit{'a forest} \Rightarrow \textit{'a forest} \Rightarrow \textit{'a forest}$$

$$\begin{array}{l}
\textit{merge}\ ts_1\ [] = ts_1 \\
\textit{merge}\ []\ ts_2 = ts_2 \\
\textit{merge}\ (t_1\ \#\ ts_1 =: f_1)\ (t_2\ \#\ ts_2 =: f_2) \\
= (\textbf{if}\ \textit{rank}\ t_1 < \textit{rank}\ t_2\ \textbf{then}\ t_1\ \#\ \textit{merge}\ ts_1\ f_2 \\
\quad\ \ \textbf{else if}\ \textit{rank}\ t_2 < \textit{rank}\ t_1\ \textbf{then}\ t_2\ \#\ \textit{merge}\ f_1\ ts_2 \\
\qquad\quad \textbf{else}\ \textit{ins\_tree}\ (\textit{link}\ t_1\ t_2)\ (\textit{merge}\ ts_1\ ts_2))
\end{array}$$

The *merge* function can be regarded as an algorithm for adding two sparse binary numbers. This intuition is explored in Exercise 17.2.

We show that the merge operation preserves the invariant and adds the elements:

$$\textit{invar}\ ts_1 \wedge \textit{invar}\ ts_2 \longrightarrow \textit{invar}\ (\textit{merge}\ ts_1\ ts_2)$$

$$\textit{mset\_forest}\ (\textit{merge}\ ts_1\ ts_2) = \textit{mset\_forest}\ ts_1 + \textit{mset\_forest}\ ts_2$$

The proof is straightforward, except for preservation of the invariant. We first show that merging two forests does not decrease the lowest rank in these forests. This ensures that prepending the head with smaller rank to the recursive merger of the remaining forests results in a sorted forest. Moreover, when we link two forests of equal rank, this ensures that the rank of t' is less or equal to the ranks of the trees in ts' (for t' and ts' see above), as required by the *ins_tree* function. We phrase this property as preservation of lower rank bounds, i.e. a lower rank bound of both forests is still a lower bound for the merged forest:

$$t' \in \textit{set}\ (\textit{merge}\ ts_1\ ts_2) \wedge (\forall t_{12}{\in}\textit{set}\ ts_1 \cup \textit{set}\ ts_2.\ \textit{rank}\ t < \textit{rank}\ t_{12}) \longrightarrow \\ \textit{rank}\ t < \textit{rank}\ t'$$

The proof is by straightforward induction, relying on an analogous bounding lemma for *ins_tree*.

17.2.3 Finding a Minimal Element

For a binomial tree, the root node always contains a minimal element. Unfortunately, there is no such property for the whole forest—the minimal element may be at the root of any tree in the forest. To get a minimal element from a non-empty forest, we look at all root nodes:

```
get_min :: 'a forest ⇒ 'a

get_min [t] = root t
get_min (t # ts) = min (root t) (get_min ts)
```

Correctness of this operation is proved by a simple induction:

$$\mathit{mset\_forest}\ ts \neq \{\!\{\}\!\} \wedge \mathit{invar}\ ts \longrightarrow \mathit{get\_min}\ ts = \mathit{Min\_mset}\ (\mathit{mset\_forest}\ ts)$$

17.2.4 Deleting a Minimal Element

To delete a minimal element, we first need to find one and then remove it. Removing the root node of a tree with rank r leaves us with a list of its children, which are binomial trees of ranks $r - 1, \ldots, 0$. Reversing this list yields a valid binomial forest, which we merge with the remaining trees in the original forest:

```
del_min :: 'a forest ⇒ 'a forest

del_min ts
= (case get_min_rest ts of (⟨_, _, ts₁⟩, ts₂) ⇒ merge (itrev ts₁ []) ts₂)
```

We use *itrev* for efficiency reasons, as explained in Section 1.5.1. The auxiliary function *get_min_rest* splits a forest into a tree with minimal root element and the remaining trees.

```
get_min_rest :: 'a forest ⇒ 'a tree × 'a forest

get_min_rest [t] = (t, [])
get_min_rest (t # ts)
= (let (t', ts') = get_min_rest ts
   in if root t ≤ root t' then (t, ts) else (t', t # ts'))
```

We prove that, for a non-empty heap, *del_min* preserves the invariant and deletes the minimal element:

$$ts \neq [] \wedge \mathit{invar}\ ts \longrightarrow \mathit{invar}\ (\mathit{del\_min}\ ts)$$

$$ts \neq [] \longrightarrow \mathit{mset\_forest}\ ts = \mathit{mset\_forest}\ (\mathit{del\_min}\ ts) + \{\!\{\mathit{get\_min}\ ts\}\!\}$$

The proof of the multiset proposition is straightforward. For invariant preservation, the key is to show that *get_min_rest* preserves the invariants:

$$\mathit{get\_min\_rest}\ ts = (t',\ ts') \wedge ts \neq [] \wedge \mathit{invar}\ ts \longrightarrow \mathit{bheap}\ t'$$
$$\mathit{get\_min\_rest}\ ts = (t',\ ts') \wedge ts \neq [] \wedge \mathit{invar}\ ts \longrightarrow \mathit{invar}\ ts'$$

To show that we actually remove a minimal element, we show that *get_min_rest* selects a tree with the same root as *get_min*:

$$ts \neq [] \wedge \mathit{get\_min\_rest}\ ts = (t',\ ts') \longrightarrow \mathit{root}\ t' = \mathit{get\_min}\ ts$$

17.3 Running Time

The running time functions are shown in Appendix B.7. Intuitively, the operations are linear in the length of the forest, which in turn is logarithmic in the number of elements (see Section 17.1).

The running time analysis for *insert* is straightforward. The running time is dominated by *ins_tree*. In the worst case, it iterates over the whole heap, taking constant time per iteration. By straightforward induction, we show

$$T_{\mathit{ins\_tree}}\ t\ ts \leq |ts| + 1$$

and thus

$$\mathit{invar}\ ts \longrightarrow T_{\mathit{insert}}\ x\ ts \leq \mathit{lg}\ (|\mathit{mset\_forest}\ ts| + 1) + 1$$

The running time analysis for *merge* is more interesting. In each call, we need constant time to compare the ranks. However, if the ranks are equal, we link the trees and insert them into the merger of the remaining forests. In the worst case, this takes time linear in the length of the merger. A naive analysis would estimate $|\mathit{merge}\ ts_1\ ts_2| \leq |ts_1| + |ts_2|$, and thus yield a quadratic running time in $|ts_1| + |ts_2|$.

However, we can do better: we observe that every link operation in *ins_tree* reduces the number of trees in the forest. Thus, over the whole merge, we can only have linearly many link operations in $|ts_1| + |ts_2|$.

To formalize this idea, we estimate the running time of *ins_tree* and *merge* together with the length of the result:

$$T_{\mathit{ins\_tree}}\ t\ ts + |\mathit{ins\_tree}\ t\ ts| = 2 + |ts|$$
$$T_{\mathit{merge}}\ ts_1\ ts_2 + |\mathit{merge}\ ts_1\ ts_2| \leq 2 \cdot (|ts_1| + |ts_2|) + 1$$

Both estimates can be proved by straightforward induction, and from the second estimate we easily derive a bound for *merge*:

$$\begin{aligned}&\mathit{invar}\ ts_1 \wedge \mathit{invar}\ ts_2 \longrightarrow\\ &T_{\mathit{merge}}\ ts_1\ ts_2 \leq 4 \cdot \mathit{lg}\ (|\mathit{mset\_forest}\ ts_1| + |\mathit{mset\_forest}\ ts_2| + 1) + 1\end{aligned}$$

From the bound for *merge* and (17.1) we can prove a bound for *del_min*:

$$\mathit{invar}\ ts \wedge ts \neq [] \longrightarrow T_{\mathit{del\_min}}\ ts \leq 6 \cdot \mathit{lg}\ (|\mathit{mset\_forest}\ ts| + 1) + 2$$

17.4 Exercises

Exercise 17.1. A node in a tree is on level n if it is n edges away from the root. Define a function *nol* :: *nat* ⇒ *'a tree* ⇒ *nat* such that *nol n t* is the number of nodes on level n in tree t and show that a binomial tree of rank r has $\binom{r}{l}$ nodes on level l. In Isabelle, $\binom{r}{l}$ is written *r choose l* and thus you should prove

$$\mathit{btree}\ t \longrightarrow \mathit{nol}\ l\ t = \mathit{rank}\ t\ \mathit{choose}\ l$$

Hint: You might want to prove separately that

$$\sum_{i=0}^{i<r} \binom{i}{n} = \binom{r}{n+1}$$

Exercise 17.2. Sparse binary numbers represent a binary number by a list of the positions of set bits, sorted in ascending order. Thus, the list [1, 3, 4] represents the number 11010. In general, $[p_1, \ldots, p_n]$ represents $2^{p_1} + \cdots + 2^{p_n}$.

Implement sparse binary numbers in Isabelle, using the type *nat list*.

1. Define a function *invar_sn* :: *nat list* ⇒ *bool* that checks for strictly ascending bit positions, a function *num_of* :: *nat list* ⇒ *nat* that converts a sparse binary number to a natural number, and a function *add* :: *nat list* ⇒ *nat list* ⇒ *nat list* to add sparse binary numbers.
2. Show that *add* preserves the invariant and actually performs addition as far as *num_of* is concerned.
3. Define a running time function for *add* and show that it is linear in the list lengths.

Hint: The bit positions in sparse binary numbers are analogous to binomial trees of a certain rank in a binomial forest. The *add* function should be implemented similarly to the *merge* function, using a *carry* function to insert a bit position into a number (similar to *ins_tree*). Correctness and running time can be proved similarly.

Chapter Notes

The binomial priority queue (often called **binomial heap**) was invented by Vuillemin [1978]. Functional implementations were given by King [1994] and Okasaki [1998]. A functional implementation was verified by Meis et al. [2010], a Java implementation by Müller [2018].

Part IV

Advanced Design and Analysis Techniques

18 Dynamic Programming

Simon Wimmer

You probably have seen this function before:

$$\mathit{fib} :: \mathit{nat} \Rightarrow \mathit{nat}$$

$$\mathit{fib}\ 0 = 0$$
$$\mathit{fib}\ 1 = 1$$
$$\mathit{fib}\ (n + 2) = \mathit{fib}\ (n + 1) + \mathit{fib}\ n$$

It computes the well-known Fibonacci numbers. You may also have noticed that calculating *fib* 50 already causes quite some stress for your computer and there is no hope for *fib* 500 to ever return a result.

This is quite unfortunate considering that there is a very simple imperative program to compute these numbers efficiently:

```
int fib(n) {
  int a = 0;
  int b = 1;
  for (i in 1..n) {
    int temp = b;
    b = a + b;
    a = temp;
  }
  return a;
}
```

So we seem to be caught in an adverse situation here: either we use a clear and elegant definition of *fib* or we get an efficient but convoluted implementation of *fib*. Admittedly, we could just prove that both formulations are the same function, and use whichever one is more suited for the task at hand. For *fib*, of course, it is trivial to define a functional analogue of the imperative program and to prove its correctness.

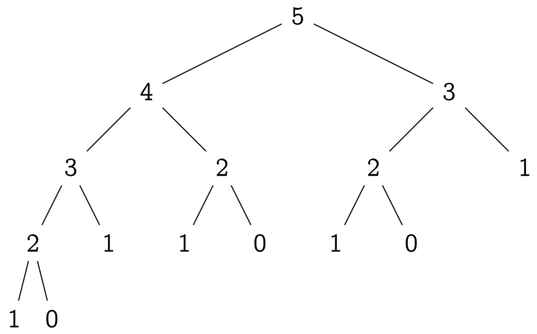


Figure 18.1 Tree of the recursive call structure for *fib* 5

However, doing this for all recursive functions we would like to define is tedious. Instead, this chapter will sketch a recipe that allows to define such recursive functions in the natural way, while still getting an efficient implementation “for free”.

In the following, the Fibonacci function will serve as a simple example on which we can illustrate the idea. Next, we will show how to prove the correctness of the efficient implementation in an efficient way. Subsequently, we will discuss further details of the approach and how it can be applied beyond *fib*. The chapter closes with the study of two famous (and archetypical) dynamic programming algorithms: the Bellman-Ford algorithm for finding shortest paths in weighted graphs and an algorithm due to Knuth for computing optimal binary search trees.

18.1 Memoization

Let us consider the tree of recursive calls that are issued when computing *fib* 5 in Figure 18.1. We can see that the subtree for *fib* 3 is computed twice, and that the subtree for *fib* 2 is even computed three times. How can we avoid these repeated computations? A common solution is **memoization**: we store previous computation results in some kind of memory and consult it to potentially recall a memoized result before issuing another recursive computation.

Below you see a simple memoizing version of *fib* that implements the memory as a map of type $nat \rightharpoonup nat$ (see Section 6.4 for the notation):

$$\mathit{fib}_1 :: nat \Rightarrow (nat \rightharpoonup nat) \Rightarrow nat \times (nat \rightharpoonup nat)$$

$$\mathit{fib}_1\ 0\ m = (0,\ m(0 \mapsto 0))$$
$$\mathit{fib}_1\ 1\ m = (1,\ m(1 \mapsto 1))$$

```
fib_1 (n + 2) m
= (let (i, m) = case m n of None ⇒ fib_1 n m | Some i ⇒ (i, m);
       (j, m) =
         case m (n + 1) of None ⇒ fib_1 (n + 1) m | Some j ⇒ (j, m)
    in (i + j, m(n + 2 ↦ i + j)))
```

And indeed, we can ask Isabelle to compute (via the **value** command) *fib*$_1$ 50 or even *fib*$_1$ 500 and we get the result within a split second.

However, we are not yet happy with this code. Carrying the memory around means a lot of additional weight for the definition of *fib*$_1$, and proving that this function computes the same value as *fib* is not completely trivial (how would you approach this?). Let us streamline the definition first by pulling out the reading and writing of memory into a function *memo* (for a type $'k$ of keys and a type $'v$ of values):

```
memo ::
  'k ⇒ (('k ⇀ 'v) ⇒ 'v × ('k ⇀ 'v))
        ⇒ ('k ⇀ 'v) ⇒ 'v × ('k ⇀ 'v)
memo k f m
= (case m k of None ⇒ let (v, m) = f m in (v, m(k ↦ v))
    | Some v ⇒ (v, m))

fib_2 :: nat ⇒ (nat ⇀ nat) ⇒ nat × (nat ⇀ nat)
fib_2 0 = memo 0 (λm. (0, m))
fib_2 1 = memo 1 (λm. (1, m))
fib_2 (n + 2)
= memo (n + 2)
    (λm. let (i, m) = fib_2 n m;
             (j, m) = fib_2 (n + 1) m
          in (i + j, m))
```

This already looks a lot more like the original definition but it still has one problem: we have to thread the memory through the program explicitly. This can become rather tedious for more complicated programs, and deviates from the original shape of the program, complicating the proofs.

18.1.1 Enter the Monad

Let us examine the type of fib_2 more closely. We can read it as the type of a function that, given a natural number, returns a **computation**. Given an initial memory, it computes a pair of a result and an updated memory. We can capture this notion of "stateful" computations in a data type:

```
datatype ('s, 'a) state = State ('s ⇒ 'a × 's)
```

A value of type *('s, 'a) state* represents a stateful computation that returns a result of type *'a* and operates on states of type *'s*. The constant *run_state* forces the evaluation of a computation starting from some initial state:

```
run_state :: ('s, 'a) state ⇒ 's ⇒ 'a × 's
run_state (State f) s = f s
```

The advantage of this definition may not seem immediate. Its value only starts to show when we see how it allows us to *chain* stateful computations. To do so, we only need to define two constants: *return* to pack up a result in a computation, and *bind* to chain two computations after each other.

```
return :: 'a ⇒ ('s, 'a) state
return x = State (λs. (x, s))

bind :: ('s, 'a) state ⇒ ('a ⇒ ('s, 'b) state) ⇒ ('s, 'b) state
bind a f = State (λs. let (x, s) = run_state a s in run_state (f x) s)
```

We add a little syntax on top and write $\langle\!\langle x \rangle\!\rangle$ for *return* x, and $a \gg\!= f$ instead of *bind* $a\ f$. The "identity" computation $\langle\!\langle x \rangle\!\rangle$ simply leaves the given state unchanged and produces x as a result. The chained computation $a \gg\!= f$ starts with some state s, runs a on it to produce a pair of a result x and a new state s', and then evaluates $f\ x$ to produce another computation that is run on s'.

We have now seen how to pass state around but we are not yet able to interact with it. For this purpose we define *get* and *set* to retrieve and update the current state, respectively:

```
get :: ('s, 's) state
get = State (λs. (s, s))

set :: 's ⇒ ('s, unit) state
set s' = State (λ_. ((), s'))
```

Let us reformulate fib_2 with the help of these concepts:

```
memo1 :: 'k ⇒ ('k ⇀ 'v, 'v) state ⇒ ('k ⇀ 'v, 'v) state
memo1 k a
= get ≫= (λm. case m k of
                None ⇒ a ≫= (λv. set (m(k ↦ v)) ≫= (λ_. ⟪v⟫)) |
                Some x ⇒ ⟪x⟫)

fib3 :: nat ⇒ (nat ⇀ nat, nat) state
fib3 0 = ⟪0⟫
fib3 1 = ⟪1⟫
fib3 (n + 2)
= memo1 (n + 2) (fib3 n ≫= (λi. fib3 (n + 1) ≫= (λj. ⟪i + j⟫)))
```

Can you see how we have managed to hide the whole handling of state behind the scenes? The only explicit interaction with the state is now happening inside of memo_1. This is sensible as this is the only place where we really want to recall a memoized result or to write a new value to memory.

While this is great, we still want to polish the definition further: the syntactic structure of the last case of fib_3 still does not match *fib* exactly. To this end, we lift function application $f\ x$ to the state monad:

```
(.) :: ('s, 'a ⇒ ('s, 'b) state) state ⇒ ('s, 'a) state ⇒ ('s, 'b) state
f_m . x_m = (f_m ≫= (λf. x_m ≫= (λx. f x)))
```

We can now spell out our final memoizing version of *fib* where (.) replaces ordinary function applications in the original definition:

$$\mathit{fib}_4 :: \mathit{nat} \Rightarrow (\mathit{nat} \rightharpoonup \mathit{nat},\ \mathit{nat})\ \mathit{state}$$
$$\mathit{fib}_4\ 0 = \langle\!\langle 0 \rangle\!\rangle$$
$$\mathit{fib}_4\ 1 = \langle\!\langle 1 \rangle\!\rangle$$
$$\mathit{fib}_4\ (n + 2)$$
$$= \mathit{memo}_1\ (n + 2)\ (\langle\!\langle \lambda i.\ \langle\!\langle \lambda j.\ \langle\!\langle i + j \rangle\!\rangle \rangle\!\rangle \rangle\!\rangle\ .\ (\mathit{fib}_4\ n)\ .\ (\mathit{fib}_4\ (n + 1)))$$

You may wonder why we added that many additional computations in this last step. On the one hand, we have gained the advantage that we can now closely follow the syntactic structure of *fib* to prove that fib_4 is correct (notwithstanding that memo_1 will need a special treatment, of course). On the other hand, we can remove most of these additional computations in a final post-processing step.

18.1.2 Memoization and Dynamic Programming

Let us recap what we have seen so far in this chapter. We noticed that the naive recursive formulation of the Fibonacci numbers leads to a highly inefficient implementation. We then showed how to work around this problem by using memoization to obtain a structurally similar but efficient implementation. After all this, you may wonder why this chapter is entitled *Dynamic Programming* and not *Memoization*.

Dynamic programming is based on two main principles. First, to find an optimal solution for a problem by computing it from optimal solutions for "smaller" instances of the same problem, i.e. *recursion*. Second, to *memoize* these solutions for smaller problems in, e.g. a table. Thus we could be bold and state:

dynamic programming = recursion + memoization

A common objection to this equation would be that memoization should be distinguished from **tabulation**. In this view, the former only computes "necessary" solutions for smaller sub-problems, while the latter just "blindly" builds solutions for sub-problems of increasing size, many of which might be unnecessary. The benefit of tabulation could be increased performance, for instance due to improved caching. We believe that this distinction is largely irrelevant to our approach. First, in this book we focus on asymptotically efficient solutions, not constant-factor optimizations. Second, in many dynamic programming algorithms memoization would actually compute solutions for the same set of sub-problems as tabulation does. No matter which of the two approaches is used in the implementation, the hard part is to come up with a recursive solution that can efficiently make use of sub-problems in the first place.

There are problems, however, where clever tabulation instead of naive memoization is necessary to achieve an asymptotically optimal solution in terms of memory consumption. One instance of this is the Bellman-Ford algorithm presented in Section

18.4. On this example, we will show that our approach is also akin to tabulation. It can easily be introduced as a final post-processing step.

Some readers may have noticed that our optimized implementations of *fib* are not really optimal as they use a map for memoization. Indeed it is possible to swap in other memory implementations as long as they provide a *lookup* and an *update* method. One can even make use of imperative data structures like arrays. Because this is not the focus of this book, the interested reader is referred to the literature that is provided at the end of this chapter. Here, we will just assume that the maps used for memoization are implemented as red-black trees (and Isabelle's code generator can be instructed to do so).

For the remainder of this chapter, we will first outline how to prove that fib_4 is correct. Then, we will sketch how to apply our approach of memoization beyond *fib*. Afterwards, we will study some prototypical examples of dynamic programming problems and show how to apply the above formula to them.

18.2 Correctness of Memoization

We now want to prove that fib_4 is correct. But what is it exactly that we want to prove? We surely want fib_4 to produce the same result as *fib* when run with an empty memory (in *this* chapter we write the empty map $\lambda\_.\ \mathit{None}$ simply as *empty*):

$$\mathit{fst}\ (\mathit{run\_state}\ (\mathit{fib}_4\ n)\ \mathit{empty}) = \mathit{fib}\ n \tag{18.1}$$

If we were to make a naive attempt at this proof, we would probably start with an induction on the computation of *fib* just to realize that the induction hypotheses are not strong enough to prove the recursion case, since they demand an empty memory. We can attempt generalization as a remedy:

$$\mathit{fst}\ (\mathit{run\_state}\ (\mathit{fib}_4\ n)\ m) = \mathit{fib}\ n$$

However, this statement does not hold anymore for every memory m.

What do we need to demand from m? It should only memoize values that are **consistent** with *fib*:

type_synonym $'a\ \mathit{mem} = (\mathit{nat} \rightharpoonup \mathit{nat},\ 'a)\ \mathit{state}$

$\mathit{cmem} :: (\mathit{nat} \rightharpoonup \mathit{nat}) \Rightarrow \mathit{bool}$

$\mathit{cmem}\ m = (\forall n{\in}\mathit{dom}\ m.\ m\ n = \mathit{Some}\ (\mathit{fib}\ n))$

$\mathit{dom} :: ('k \rightharpoonup 'v) \Rightarrow 'k\ \mathit{set}$

$\mathit{dom}\ m = \{a \mid m\ a \neq \mathit{None}\}$

Note that, from now on, we use the type $'a\ mem$ to denote *memoized* values of type $'a$ that have been "wrapped up" in our memoizing state monad. Using *cmem*, we can formulate a general notion of equivalence between a value v and its memoized version a, written $v \triangleright a$: starting from a consistent memory m, a should produce another consistent memory m', and the result v.

$$\begin{array}{l} (\triangleright) :: {'a} \Rightarrow {'a}\ mem \Rightarrow bool \\ v \triangleright a \\ = (\forall m.\ \mathit{cmem}\ m \longrightarrow \\ \qquad\quad (\mathbf{let}\ (v',\ m') = \mathit{run\_state}\ a\ m\ \mathbf{in}\ v = v' \wedge \mathit{cmem}\ m')) \end{array}$$

Thus we want to prove

$$\mathit{fib}\ n \triangleright \mathit{fib_4}\ n \tag{18.2}$$

via computation induction on n. For the base cases we need to prove statements of the form $v \triangleright \langle\!\langle v \rangle\!\rangle$, which follow trivially after unfolding the involved definitions. For the induction case, we can unfold $\mathit{fib_4}\ (n + 2)$, and get rid of $\mathit{memo_1}$ by applying the following rule (which we instantiate with $a = \mathit{fib}_4\ n$):

$$\mathit{fib}\ n \triangleright a \longrightarrow \mathit{fib}\ n \triangleright \mathit{memo_1}\ n\ a \tag{18.3}$$

For the remainder of the proof, we now want to unfold $\mathit{fib}\ (n + 2)$ and then follow the syntactic structure of $\mathit{fib_4}$ and fib in lockstep. To do so, we need to find a proof rule for function application. That is, what do we need in order to prove $f\ x \triangleright f_m\ .\ x_m$? For starters, $x \triangleright x_m$ seems reasonable to demand. But what about f and f_m? If f has type $'a \Rightarrow {'b}$, then f_m is of type $('a \Rightarrow {'b}\ mem)\ mem$. Intuitively, we want to state something along these lines:

> f_m is a memoized function that, when applied to a value x, yields a memoized value that is equivalent to $f\ x$.

This goes beyond what we can currently express with $(\triangleright)$ as $v \triangleright a$ merely states that "a is a memoized value equivalent to v". What we need is more liberty in our choice of equivalence. That is, we want to use statements $v \triangleright_R a$, with the meaning: "a is a memoized value that is related to v by R". The formal definition is analogous to $(\triangleright)$ (and $(\triangleright) = (\triangleright_{(=)})$):

$$(\cdot\triangleright\cdot) :: {}'a \Rightarrow ({}'a \Rightarrow {}'b \Rightarrow \mathit{bool}) \Rightarrow {}'b\ \mathit{mem} \Rightarrow \mathit{bool}$$

$$\begin{aligned}&v \triangleright_R s\\ &= (\forall m.\ \textsf{cmem}\ m \longrightarrow\\ &\qquad (\textbf{let}\ (v',\ m') = \textsf{run\_state}\ s\ m\ \textbf{in}\ R\ v\ v' \wedge \textsf{cmem}\ m'))\end{aligned}$$

However, we still do not have a means of expressing the second part of our sentence. To this end, we use the **function relator** $(\Rrightarrow)$:

$$(\Rrightarrow) :: ({}'a \Rightarrow {}'c \Rightarrow \mathit{bool}) \Rightarrow ({}'b \Rightarrow {}'d \Rightarrow \mathit{bool}) \Rightarrow ({}'a \Rightarrow {}'b) \Rightarrow ({}'c \Rightarrow {}'d) \Rightarrow \mathit{bool}$$

$$R \Rrightarrow S = (\lambda f\ g.\ \forall x\ y.\ R\ x\ y \longrightarrow S\ (f\ x)\ (g\ y))$$

Spelled out, we have $(R \Rrightarrow S)\ f\ g$ if for any values x and y that are related by R, the values $f\ x$ and $g\ y$ are related by S.

We can finally state a proof rule for application:

$$x \triangleright x_m \wedge f \triangleright_{(=)\ \Rrightarrow\ (\triangleright)} f_m \longrightarrow f\ x \triangleright f_m\ .\ x_m \tag{18.4}$$

In our concrete example, we apply it once to the goal

$$\textsf{fib}\ (n + 1) + \textsf{fib}\ n \triangleright \langle\!\langle \lambda a.\ \langle\!\langle \lambda b.\ \langle\!\langle a + b \rangle\!\rangle \rangle\!\rangle \rangle\!\rangle\ .\ (\textsf{fib}_4\ (n + 1))\ .\ (\textsf{fib}_4\ n)$$

solve the first premise with the induction hypotheses, and arrive at

$$(+)\ (\textsf{fib}\ (n + 1)) \triangleright_{(=)\ \Rrightarrow\ (\triangleright)} \langle\!\langle \lambda a.\ \langle\!\langle \lambda b.\ \langle\!\langle a + b \rangle\!\rangle \rangle\!\rangle \rangle\!\rangle\ .\ (\textsf{fib}_4\ (n + 1))$$

Our current rule for application (18.4) does not match this goal. Thus we need to generalize it. In addition, we need a new rule for *return*, and a rule for $(\Rrightarrow)$. To summarize, we need the following set of theorems about our consistency relation, applying them wherever they match syntactically to finish the proof of (18.2):

$$R\ x\ y \longrightarrow x \triangleright_R \langle\!\langle y \rangle\!\rangle$$

$$x \triangleright_R x_m \wedge f \triangleright_{R\ \Rrightarrow\ \triangleright_S} f_m \longrightarrow f\ x \triangleright_S f_m\ .\ x_m$$

$$(\forall x\ y.\ R\ x\ y \longrightarrow S\ (f\ x)\ (g\ y)) \longrightarrow (R \Rrightarrow S)\ f\ g$$

The theorem we aimed for initially

$$\textsf{fst}\ (\textsf{run\_state}\ (\textsf{fib}_4\ n)\ \textsf{empty}) = \textsf{fib}\ n \tag{18.1}$$

is now a trivial corollary of $\textsf{fib}\ n \triangleright \textsf{fib}_4\ n$. By reading the equation from right to left, we have an easy way to make the memoization transparent to an end-user of *fib*.

18.3 Details of Memoization*

In this section, we will look at some further details of the memoization process and sketch how it can be applied beyond *fib*. First note that our approach of memoization hinges on two rather independent components: We transform the original program to use the state monad, to thread (an *a priori* arbitrary) state through the program. Only at the call sites of recursion, we then introduce the memoization functionality by issuing lookups and updates to the memory (as implemented by memo_1). We will name this first process **monadification**. For the second component, many different memory implementations can be used, as long as we can define memo_1 and prove its characteristic theorem (18.3). For details on this, the reader is referred to the literature. Here, we want to turn our attention towards monadification.

To discuss some of the intricacies of monadification, let us first stick with *fib* for a bit longer and consider the following alternative definition (which is mathematically equivalent but not the same program):

$$\mathit{fib}\ n = (\textbf{if}\ n = 0\ \textbf{then}\ 0\ \textbf{else}\ 1 + \mathit{sum\_list}\ (\mathit{map}\ \mathit{fib}\ [0..{<}n - 1]))$$

We have not yet seen how to handle two ingredients of this program: constructs like **if**-**then**-**else** or case-combinators; and higher-order functions such as *map*.

It is quite clear how **if**-**then**-**else** can be lifted to the state monad:

$$\mathit{if}_m :: \mathit{bool\ mem} \Rightarrow \mathit{'a\ mem} \Rightarrow \mathit{'a\ mem} \Rightarrow \mathit{'a\ mem}$$
$$\mathit{if}_m\ b_m\ x_m\ y_m = b_m \gg\!= (\lambda b.\ \textbf{if}\ b\ \textbf{then}\ x_m\ \textbf{else}\ y_m)$$

By following the structure of the terms, we can also deduce a proof rule for if_m:

$$b \triangleright b_m \wedge x \triangleright_R x_m \wedge y \triangleright_R y_m \longrightarrow (\textbf{if}\ b\ \textbf{then}\ x\ \textbf{else}\ y) \triangleright_R \mathit{if}_m\ b_m\ x_m\ y_m$$

However, suppose we want to apply this proof rule to our new equation for *fib*. We will certainly need the knowledge of whether $n = 0$ to make progress in the correctness proof. Thus we make our rule more precise:

$$b \triangleright b_m \wedge (b \longrightarrow x \triangleright_R x_m) \wedge (\neg\ b \longrightarrow y \triangleright_R y_m) \longrightarrow$$
$$(\textbf{if}\ b\ \textbf{then}\ x\ \textbf{else}\ y) \triangleright_R \mathit{if}_m\ b_m\ x_m\ y_m$$

How can we lift *map* to the state monad level? Consider its defining equations:

*If you are just interested in the dynamic programming algorithms of the following sections, this section can safely be skipped on first reading.

$$
\begin{aligned}
&\mathsf{map}\ f\ []\ =\ []\\
&\mathsf{map}\ f\ (x\ \#\ xs)\ =\ f\ x\ \#\ \mathsf{map}\ f\ xs
\end{aligned}
$$

We can follow the pattern we used to monadify *fib* to monadify *map*:

$$
\begin{aligned}
&\mathsf{map}_m{}'\ f\ []\ =\ \langle\!\langle[]\rangle\!\rangle\\
&\mathsf{map}_m{}'\ f\ (x\ \#\ xs)\ =\ \langle\!\langle\lambda a.\ \langle\!\langle\lambda b.\ \langle\!\langle a\ \#\ b\rangle\!\rangle\rangle\!\rangle\rangle\!\rangle\ .\ (\langle\!\langle f\rangle\!\rangle\ .\ \langle\!\langle x\rangle\!\rangle)\ .\ (\mathsf{map}_m{}'\ f\ xs)
\end{aligned}
$$

We have obtained a function $\mathsf{map}_m{}'$ of type

$$('a \Rightarrow 'b\ mem) \Rightarrow 'a\ list \Rightarrow 'b\ list\ mem$$

This is not yet compatible with our scheme of lifting function application to (.). We need a function of type

$$(('a \Rightarrow 'b\ mem) \Rightarrow ('a\ list \Rightarrow 'b\ list\ mem)\ mem)\ mem$$

because *map* has two arguments and we need one layer of the state monad for each of its arguments. Therefore we simply define

$$\mathsf{map}_m\ =\ \langle\!\langle\lambda f.\ \langle\!\langle\mathsf{map}_m{}'\ f\rangle\!\rangle\rangle\!\rangle$$

For inductive proofs about the new definition of *fib*, we also need the knowledge that *fib* is recursively applied only to smaller values than n when computing *fib* n. That is, we need to know which values f is applied to in *map* $f\ xs$. We can encode this knowledge in a proof rule for *map*:

$$
\begin{aligned}
&xs\ =\ ys\ \wedge\ (\forall x.\ x \in \mathsf{set}\ ys \longrightarrow f\ x \triangleright_R f_m\ x) \longrightarrow\\
&\mathsf{map}\ f\ xs \triangleright_{\mathsf{list\_all2}\ R}\ \mathsf{map}_m\ .\ \langle\!\langle f_m\rangle\!\rangle\ .\ \langle\!\langle ys\rangle\!\rangle
\end{aligned}
$$

The relator *list_all2* lifts R to a pairwise relation on lists:

$$\mathsf{list\_all2}\ R\ xs\ ys\ =\ (|xs|\ =\ |ys|\ \wedge\ (\forall i{<}|xs|.\ R\ (xs\ !\ i)\ (ys\ !\ i)))$$

To summarize, here is a fully memoized version of the alternative definition of *fib*:

$$
\begin{aligned}
&\mathsf{fib}_m :: nat \Rightarrow nat\ mem\\
&\mathsf{fib}_m\ n\ =\ \mathsf{memo}_1\ n\\
&\ (\mathsf{if}_m\ \langle\!\langle n\ =\ 0\rangle\!\rangle\ \langle\!\langle 0\rangle\!\rangle\\
&\ \ (\langle\!\langle\lambda a.\ \langle\!\langle 1\ +\ a\rangle\!\rangle\rangle\!\rangle\ .\ (\langle\!\langle\lambda a.\ \langle\!\langle\mathsf{sum\_list}\ a\rangle\!\rangle\rangle\!\rangle\ .\ (\mathsf{map}_m\ .\ \langle\!\langle\mathsf{fib}_m{}'\rangle\!\rangle\ .\ \langle\!\langle[0..{<}n\ -\ 1]\rangle\!\rangle))))
\end{aligned}
$$

The correctness proof for fib_m is analogous to the one for fib_4, once we have proved the new rules discussed above.

At the end of this section, we note that the techniques that were sketched above also extend to case-combinators and other higher-order functions. Most of the machinery for monadification and the corresponding correctness proofs can be automated in Isabelle [Wimmer et al. 2018b]. Finally note that none of the techniques we used so far are specific to *fib*. The only parts that have to be adopted are the definitions of memo_1 and *cmem*. In Isabelle, this can be done by simply instantiating a locale.

This concludes the discussion of the fundamentals of our approach towards verified dynamic programming. We now turn to the study of two typical examples of dynamic programming algorithms: the Bellman-Ford algorithm and an algorithm for computing optimal binary search trees.

18.4 The Bellman-Ford Algorithm

Computing shortest paths in weighted graphs is a classic algorithmic task that we all encounter in everyday situations, such as planning the fastest route to drive from A to B. In this scenario we can view streets as edges in a graph and nodes as street crossings. Each edge is associated with a weight, e.g. the time to traverse a street. We are interested in the path from A to B with minimum weight, corresponding to the fastest route in the example. Note that in this example it is safe to assume that all edge weights are non-negative.

Some applications demand negative edge weights as well. Suppose, we transport ourselves a few years into the future, where we have an electric car that can recharge itself using solar cells while driving. If we aim for the most energy-efficient route from A to B, a very sunny route could then incur a negative edge weight.

The **Bellman-Ford algorithm** is a classic dynamic programming solution to the **single-destination shortest path problem** in graphs with negative edge weights. That is, we are given a directed graph with negative edge weights and some target vertex (called a **sink**), and we want to compute the weight of the shortest (i.e. minimum weight) paths from each vertex to the sink. Figure 18.2 shows an example of such a graph.

Formally, we will take a simple view of graphs. We assume that we are given a number of nodes numbered $0, \ldots, n$, and some sink $t \in \{0..n\}$ (thus $n = t = 4$ in the example). Edge weights are given by a function $W :: \mathit{int} \Rightarrow \mathit{int} \Rightarrow \mathit{int\ extended}$. The type *int extended* extends the integers with positive and negative infinity:

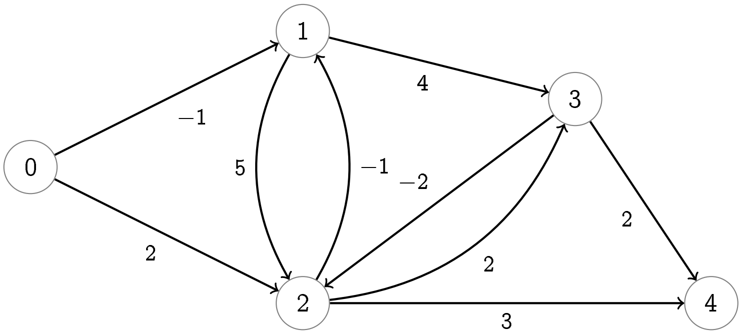


Figure 18.2 Example of a weighted directed graph

datatype *'a extended* = *Fin 'a* | ∞ | $-\infty$

We refrain from giving the explicit definition of addition and comparison on this domain, and rely on your intuition instead. A weight assignment $W\ i\ j = \infty$ means that there is no edge from i to j. The purpose of $-\infty$ will become clear later.

18.4.1 Deriving a Recursive Solution

The main idea of the algorithm is to consider paths in order of increasing length in the number of edges. In the example, we can immediately read off the weights of the shortest paths to the sink that use only one edge: only nodes 2 and 3 are directly connected to the sink, with edge weights 3 and 2, respectively; for all others the weight is infinite. How can we now calculate the minimum weight paths (to the sink) with at most two edges? For node 3, the weight of the shortest path with at most two edges is: either the weight of the path with one edge; or the weight of the edge from node 3 to node 2 plus the weight of the path with one edge from node 2 to the sink. Because $-2 + 3 = 1 \leq 2$, we get a new minimum weight of 1 for node 3. Following the same scheme, we can iteratively calculate the minimum path weights given in table 18.1.

The analysis we just ran on the example already gives us a clear intuition on all we need to deduce a dynamic program: a recursion on sub-problems, in this case to compute the weight of shortest paths with at most $i + 1$ edges from the weights of shortest paths with at most i edges. To formalize this recursion, we first define the notion of a minimum weight path from some node v to t with at most i edges, denoted as *OPT* $i\ v$:

i/v	0	1	2	3	4
0	∞	∞	∞	∞	0
1	∞	∞	3	2	0
2	5	6	3	1	0
3	5	5	3	1	0
4	4	5	3	1	0

Table 18.1 The minimum weights of paths from vertices $v = 0 \ldots 4$ to t that use at most $i = 0 \ldots 4$ edges.

$$\mathsf{OPT} :: \mathit{nat} \Rightarrow \mathit{nat} \Rightarrow \mathit{int\ extended}$$

$$\begin{aligned}&\mathsf{OPT}\ i\ v\\ &= \mathsf{Min}\ (\{\mathsf{weight}\ (v\ \#\ xs\ @\ [t])\ \mid\ |xs| + 1 \le i \wedge \mathsf{set}\ xs \subseteq \{0..n\}\} \cup\\ &\qquad \{\textbf{if}\ t = v\ \textbf{then}\ 0\ \textbf{else}\ \infty\})\end{aligned}$$

$$\mathsf{weight} :: \mathit{nat\ list} \Rightarrow \mathit{int\ extended}$$

$$\begin{aligned}&\mathsf{weight}\ [v] = 0\\ &\mathsf{weight}\ (v\ \#\ w\ \#\ xs) = W\ v\ w + \mathsf{weight}\ (w\ \#\ xs)\end{aligned}$$

If $i = 0$, things are simple:

$$\mathsf{OPT}\ 0\ v = (\textbf{if}\ t = v\ \textbf{then}\ 0\ \textbf{else}\ \infty)$$

A shortest path that constitutes $\mathsf{OPT}\ (i + 1)\ v$ uses either at most i or exactly $i + 1$ edges. That is, $\mathsf{OPT}\ (i + 1)\ v$ is either $\mathsf{OPT}\ i\ v$, or the weight of the edge from v to any of its neighbours w plus $\mathsf{OPT}\ i\ w$:

$$\mathsf{OPT}\ (i + 1)\ v = \mathsf{min}\ (\mathsf{OPT}\ i\ v)\ (\mathsf{Min}\ \{W\ v\ w + \mathsf{OPT}\ i\ w \mid w \le n\})$$

Proof. We prove this equality by proving two inequalities:

($lhs \le rhs$) For this direction, we essentially need to show that every path on the rhs is covered by the lhs, which is trivial.

($lhs \ge rhs$) We skip the cases where $\mathsf{OPT}\ (i + 1)\ v$ is trivially 0 or ∞ (i.e. where it is given by the singleton set in the definition of OPT). Thus consider some xs such that $\mathsf{OPT}\ (i + 1)\ v = \mathsf{weight}\ (v\ \#\ xs\ @\ [t])$, $|xs| \le i$, and $\mathsf{set}\ xs \subseteq \{0..n\}$. The cases where $|xs| < i$ or $i = 0$ are trivial. Otherwise, we have $\mathsf{OPT}\ (i + 1)\ v = W\ v\ (\mathsf{hd}\ xs) + \mathsf{weight}\ (xs\ @\ [t])$ by definition of weight, and $\mathsf{OPT}\ i\ (\mathsf{hd}\ xs) \le \mathsf{weight}\ (xs\ @\ [t])$ by definition of OPT. Therefore, we can show: $\mathsf{OPT}\ (i + 1)\ v \ge W\ v\ (\mathsf{hd}\ xs) + \mathsf{OPT}\ i\ (\mathsf{hd}\ xs) \ge rhs$ □

We can turn these equations into a recursive program:

```
bf :: nat ⇒ nat ⇒ int extended
bf 0 v = (if t = v then 0 else ∞)
bf (i + 1) v = min_list (bf i v # map (λw. W v w + bf i w) [0..<n + 1])
```

It is obvious that we can prove correctness of *bf* by induction:

$$\textit{bf}\ i\ v = \textit{OPT}\ i\ v$$

18.4.2 Negative Cycles

Have we solved the initial problem now? The answer is "not quite" because we have ignored one additional complication. Consider our example Table 18.1 again. The table stops at path length five because no shorter paths with more edges exist. For this example, five corresponds to the number of nodes, which bounds the length of the longest **simple path** (= without repeated nodes). However, is it the case that we will never find shorter non-simple paths in other graphs? The answer is "no". If a graph contains a **negative reaching cycle**, i.e. a cycle with a negative sum of edge weights from which the sink is reachable, then we can use it arbitrarily often to find shorter and shorter paths.

Luckily, we can use the Bellman-Ford algorithm to detect this situation by examining the relationship of $\textit{OPT}\ n$ and $\textit{OPT}\ (n + 1)$. The following proposition summarizes the key insight:

> The graph contains a negative reaching cycle if and only if there exists a $v \le n$ such that $\textit{OPT}\ (n + 1)\ v < \textit{OPT}\ n\ v$

Proof. If there is no negative reaching cycle, then all shortest paths are either simple or contain superfluous cycles of weight 0. Thus, we have $\textit{OPT}\ (n + 1)\ v = \textit{OPT}\ n\ v$ for all $v \le n$.

Otherwise, there is a negative reaching cycle $ys = a\ \#\ xs\ @\ [a]$ with $\textit{weight}\ ys < 0$. Working towards a contradiction, assume that $\textit{OPT}\ n\ v \le \textit{OPT}\ (n + 1)\ v$ for all $v \le n$. Using the recursion we proved above, this implies $\textit{OPT}\ n\ v \le W\ v\ u + \textit{OPT}\ n\ u$ for all u, $v \le n$. By applying this inequality to the nodes in $a\ \#\ xs$, we can prove the inequality

$$\textit{sum\_list}\ (\textit{map}\ (\textit{OPT}\ n)\ ys) \le \textit{sum\_list}\ (\textit{map}\ (\textit{OPT}\ n)\ ys) + \textit{weight}\ ys$$

This implies $\textit{weight}\ ys \ge 0$, which yields the contradiction. □

This means we can use *bf* to detect the existence of negative reaching cycles by computing one more round, i.e. $\textit{bf}\ (n + 1)\ v$ for all v. If nothing changes in this step, we know that there are no negative reaching cycles and that *bf* n correctly represents the shortest path weights. Otherwise, there has to be a negative reaching cycle.

Finally, we can use memoization to obtain an efficient implementation that solves the single-destination shortest path problem. Applying our memoization technique from above, we first obtain a memoizing version *bf*$_m$ of *bf*. We then define the following program:

```
bellman_ford ::
  ((nat × nat, int extended) mapping, int extended list option) state
bellman_ford
= iter_bf (n, n) ≫=
   (λ_. map_m' (bf_m n) [0..<n + 1] ≫=
         (λxs. map_m' (bf_m (n + 1)) [0..<n + 1] ≫=
                (λys. ⟪if xs = ys then Some xs else None⟫)))
```

Here, *iter_bf* (n, n) just computes the values from *bf*$_m$ 0 0 to *bf*$_m$ n n in a row-by-row manner, storing them in a table (where $('a, 'b)$ *mapping* is essentially $'a \rightharpoonup 'b$). Using the reasoning principles that were described above (for *fib*), we can then prove that *bellman_ford* indeed solves its intended task correctly (*shortest* v is the length of the shortest path from v to t):

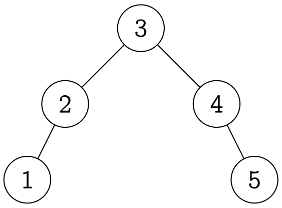
```
(∀i≤n. ∀j≤n. −∞ < W i j) ⟶
fst (run_state bellman_ford empty)
= (if contains_negative_reaching_cycle then None
   else Some (map shortest [0..<n + 1]))
```

Here, *shortest* is defined analogously to OPT but for paths of unbounded length.

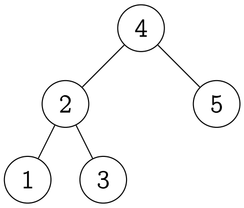
18.5 Optimal Binary Search Trees

In this book, we have studied various tree data structures that guarantee logarithmic running time bounds for lookup and update operations. These bounds were worst-case and did not take into account any information about the actual sequence of queries. In this section, instead, we focus on BSTs that minimize the amount of work when the distribution of keys in a sequence of queries is known in advance.

More formally, we study the following problem. We are given a list $[i..j]$ of integers ranging from i to j and a function p :: *int* ⇒ *nat* that maps each key in the range to a frequency with which this key is searched for. Our goal is to construct a BST that minimizes the expected number of comparisons when presented with a sequence of lookup operations for keys in the range $[i..j]$ that adhere to the distribution given by p. As an example, consider the range $[1..5]$ with frequencies $[10, 30, 15, 25, 20]$. This tree



incurs an expected value of 2.15 comparison operations. However, the minimal expected value is 2 and is achieved by this tree:



Our task is equivalent to minimizing the **weighted path length** (or *cost*). The weighted path length is the sum of the frequencies of each node in the tree multiplied by its depth in the tree. The following definition avoids the notion of depth:

$$\begin{aligned}
&\mathsf{cost} :: \mathit{int\ tree} \Rightarrow \mathit{nat}\\
&\mathsf{cost}\ \langle\rangle = 0\\
&\mathsf{cost}\ \langle l,\ k,\ r\rangle\\
&= (\textstyle\sum_{k \in \mathsf{set\_tree}\ l}\ p\ k) + \mathsf{cost}\ l + p\ k + \mathsf{cost}\ r + (\textstyle\sum_{k \in \mathsf{set\_tree}\ r}\ p\ k)
\end{aligned}$$

To come up with a dynamic programming solution, we must find a way to subdivide the problem.

18.5.1 Deriving a Recursive Solution

One way to subdivide the problem is to subdivide the interval $[i..j]$. This motivates the following definition, which generalizes *cost*:

$$\begin{aligned}
&\mathsf{wpl}\ W\ i\ j\ \langle\rangle = 0\\
&\mathsf{wpl}\ W\ i\ j\ \langle l,\ k,\ r\rangle = \mathsf{wpl}\ W\ i\ (k - 1)\ l + \mathsf{wpl}\ W\ (k + 1)\ j\ r + W\ i\ j
\end{aligned}$$

When setting $W\ i\ j = (\sum_{k\ =\ i}^{j}\ p\ k)$, it is easy to see that $\mathsf{wpl}\ W\ i\ j$ is just a reformulation of $\mathsf{cost}\ t$:

$$\mathsf{inorder}\ t = [i..j] \longrightarrow \mathsf{wpl}\ W\ i\ j\ t = \mathsf{cost}\ t$$

We can actually forget about the original frequencies p and just optimize $\mathsf{wpl}\ W\ i\ j$ for some fixed weight function $W :: int \Rightarrow int \Rightarrow nat$. Therefore, in the remainder, we will assume W to be known in the context and just write $\mathsf{wpl}\ i\ j$.

The key insight into the problem is that subtrees of optimal BSTs are also optimal. The left and right subtrees of the root must be optimal, since if we could improve either one, we would also get a better tree for the complete range of keys.

Formally, the BST t that contains the keys $[i..j]$ and minimizes $\mathsf{wpl}\ i\ j\ t$ has some root k with $[i..j] = [i..k - 1]\ @\ k\ \#\ [j + 1..k]$. Its left and right subtrees need to be minimal again, i.e. minimize $\mathsf{wpl}\ i\ (k - 1)$ and $\mathsf{wpl}\ (k + 1)\ j$. This yields the following recursive functions for computing the minimal weighted path length (*min_wpl*) and a corresponding BST (*opt_bst*):

```
min_wpl :: int ⇒ int ⇒ nat
min_wpl i j
= (if j < i then 0
   else min_list
          (map (λk. min_wpl i (k − 1) + min_wpl (k + 1) j + W i j) [i..j]))

opt_bst :: int ⇒ int ⇒ int tree
opt_bst i j
= (if j < i then ⟨⟩
   else argmin (wpl i j)
          (map (λk. ⟨opt_bst i (k − 1), k, opt_bst (k + 1) j⟩) [i..j]))
```

Here $\mathsf{argmin}\ f\ xs$ returns the rightmost $x \in \mathsf{set}\ xs$ such that $f\ x$ is minimal among xs (i.e. $f\ x \le f\ y$ for all $y \in set\ xs$).

To prove that *min_wpl* and *opt_bst* are correct, we want to show two propositions: $\mathsf{min\_wpl}\ i\ j$ should be a lower bound of $\mathsf{wpl}\ i\ j\ t$ for any search tree t for $[i..j]$, and $\mathsf{min\_wpl}\ i\ j$ should correspond to the weight of an actual search tree, namely $\mathsf{opt\_bst}\ i\ j$. Formally, we prove the following propositions:

$$\mathsf{inorder}\ t = [i..j] \longrightarrow \mathsf{min\_wpl}\ i\ j \le \mathsf{wpl}\ i\ j\ t$$

$$\mathsf{inorder}\ (\mathsf{opt\_bst}\ i\ j) = [i..j]$$

$$\mathsf{wpl}\ i\ j\ (\mathsf{opt\_bst}\ i\ j) = \mathsf{min\_wpl}\ i\ j$$

The three propositions are easily proved by computation induction on *wpl*, *opt_bst* and *min_wpl*, respectively.

When setting $W = \mathit{W}$, we can derive the following correctness theorems referring to the original problem:

$$\mathit{inorder}\ t = [i..j] \longrightarrow \mathit{min\_wpl}\ \mathit{W}\ i\ j \leq \mathit{cost}\ t$$

$$\mathit{cost}\ (\mathit{opt\_bst}\ \mathit{W}\ i\ j) = \mathit{min\_wpl}\ \mathit{W}\ i\ j$$

18.5.2 Memoization

We can apply the memoization techniques that were discussed above to efficiently compute *min_wpl* and *opt_bst*. The only remaining caveat is that *W* also needs to be computed efficiently from the distribution p. If we just use the defining equality $\mathit{W}\ i\ j = (\sum_{k\ =\ i}^{j} p\ k)$, the computation of *W* is unnecessarily costly. Another way is to memoize *W* itself, using the following recursion:

$$\mathit{W}\ i\ j = (\mathbf{if}\ i \leq j\ \mathbf{then}\ \mathit{W}\ i\ (j - 1) + p\ j\ \mathbf{else}\ 0)$$

This yields a memoizing version $\mathit{W_m}'$ and a theorem that connects it to *W*:

$$\mathit{W}\ i\ j \triangleright \mathit{W_m}'\ i\ j$$

We can now iterate $\mathit{W_m}'\ i\ n$ for $i = 0 \ldots n$ to pre-compute all relevant values of $\mathit{W}\ i\ j$:

$$\mathit{W_c}\ n = \mathit{snd}\ (\mathit{run\_state}\ (\mathit{map_m}'\ (\lambda i.\ \mathit{W_m}'\ i\ n)\ [0..n])\ \mathit{empty})$$

Using the correctness theorem for $\mathit{map_m}'$ from above, it can easily be shown that this yields a consistent memory:

$$\mathit{cmem}\ (\mathit{W_c}\ n)$$

We can show the following equation for computing *W*

$$\mathit{W}\ i\ j = (\mathbf{case}\ (\mathit{W_c}\ n)\ (i,\ j)\ \mathbf{of}\ \mathit{None} \Rightarrow \mathit{W}\ i\ j \mid \mathit{Some}\ x \Rightarrow x)$$

Note that the *None* branch will only be triggered when indices outside of $0 \ldots n$ are accessed. Finally, we can use W_c to pass the pre-computed values of *W* to *opt_bst*:

$$\mathit{opt\_bst}' :: \mathit{int} \Rightarrow \mathit{int} \Rightarrow \mathit{int\ tree}$$

$$\begin{array}{l}\mathit{opt\_bst}'\ i\ j \equiv \\ \mathbf{let}\ M = \mathit{W_c}\ j;\ W = \lambda i\ j.\ \mathbf{case}\ M\ (i,\ j)\ \mathbf{of}\ \mathit{None} \Rightarrow \mathit{W}\ i\ j \mid \mathit{Some}\ x \Rightarrow x \\ \mathbf{in}\ \mathit{opt\_bst}\ W\ i\ j\end{array}$$

18.5.3 Optimizing the Recursion

While we have applied some trickery to obtain an efficient implementation of the simple dynamic programming algorithm expressed by *opt_bst*, we still have not arrived at the solution that is currently known to be most efficient. The most efficient known algorithm to compute optimal BSTs due to Knuth [1971] is a slight variation of *opt_bst* and relies on the following observation.

Let $\mathit{R}\ i\ j$ denote the maximal root of any optimal BST for $[i..j]$:

$$\mathit{R}\ i\ j = \mathit{argmin}\ (\lambda k.\ w\ i\ j + \mathit{min\_wpl}\ i\ (k - 1) + \mathit{min\_wpl}\ (k + 1)\ j)\ [i..j]$$

It can be shown that $\mathit{R}\ i\ j$ is bounded by $\mathit{R}\ i\ (j - 1)$ and $\mathit{R}\ (i + 1)\ j$:

$$i < j \longrightarrow \mathit{R}\ i\ (j - 1) \leq \mathit{R}\ i\ j \wedge \mathit{R}\ i\ j \leq \mathit{R}\ (i + 1)\ j$$

The proof of this fact is rather involved and the details can be found in the references provided at the end of this section.

With this knowledge, we can make the following optimization to *opt_bst*:

```
opt_bst₂ :: int ⇒ int ⇒ int tree
opt_bst₂ i j
= (if j < i then ⟨⟩
   else if i = j then ⟨⟨⟩, i, ⟨⟩⟩
        else let left = root (opt_bst₂ i (j − 1));
                 right = root (opt_bst₂ (i + 1) j)
             in argmin (wpl i j)
                 (map (λk. ⟨opt_bst₂ i (k − 1), k, opt_bst₂ (k + 1) j⟩)
                   [left..right]))
```

You may wonder whether this change really results in an asymptotic runtime improvement. Indeed, it can be shown that it improves the algorithm's runtime by a factor of $O(n)$. For a fixed search tree size $d = i - j$, the total number of recursive computations is given by the following telescoping series:

$$\begin{aligned}
&d \leq n \longrightarrow \\
&(\textstyle\sum_{j = d}^{n} \textbf{let } i = j - d \textbf{ in } \mathit{R}\ (i + 1)\ j - \mathit{R}\ i\ (j - 1) + 1) \\
&= \mathit{R}\ (n - d + 1)\ n - \mathit{R}\ 0\ (d - 1) + n - d + 1
\end{aligned}$$

This quantity is bounded by $2 \cdot n$, which implies that the overall number of recursive calls is bounded by $O(n^2)$.

Chapter Notes

The original $O(n^2)$ algorithm for optimal BSTs is due to Knuth [1971]. Yao [1980] later explained this optimization more elegantly in her framework of "quadrilateral inequalities". Nipkow and Somogyi [2018] follow Yao's approach in their Isabelle formalization, on which the last subsection of this chapter is based. Chapter 26 studies a related but simpler problem, the construction of an optimal binary tree, without the ordering requirement. That problem can be solved efficiently with a greedy algorithm.

The other parts of this chapter are based on a paper by Wimmer et al. [2018b] and its accompanying Isabelle formalization [Wimmer et al. 2018a]. The formalization also contains further examples of dynamic programming algorithms, including solutions for the Knapsack and the minimum edit distance problems, and the CYK algorithm.

19 Amortized Analysis ↗

Tobias Nipkow

Consider a k-bit binary counter and a sequence of increment (by 1) operations on it where each one starts from the least significant bit and keeps flipping the 1s until a 0 is encountered (and flipped). Thus the worst-case running time of an increment is $O(k)$ and a sequence of n increments takes time $O(nk)$. However, this analysis is very coarse: in a sequence of increments there are many much faster ones (for half of them the least significant bit is 0!). It turns out that a sequence of n increments takes time $O(n)$. Thus the average running time of each increment is $O(1)$. Amortized analysis is the analysis of the running time of a sequence of operations on some data structure by upper-bounding the average running time of each operation.

As the example of the binary counter shows, the amortized running time for a single call of an operation can be much better than the worst-case time. Thus amortized analysis is unsuitable in a real-time context where worst-case bounds on every call of an operation are required.

Amortized analysis of some data structure is valid if the user of that data structure never accesses old versions of the data structure (although in a functional language one could). The binary counter shows why that invalidates amortized analysis: start from 0, increment the counter until all bits are 1, then increment that counter value again and again, without destroying it. Each of those increments takes time $O(k)$ and you can do that as often as you like, thus subverting the analysis. In an imperative language you can easily avoid this “abuse” by making the data structure stateful: every operation modifies the state of the data structure. This shows that amortized analysis has an imperative flavour. In a purely functional language, monads can be used to restrict access to the latest version of a data structure.

19.1 The Potential Method

The **potential method** is a particular technique for amortized analysis. The key idea is to define a **potential function** Φ from the data structure to non-negative numbers. The potential of the data structure is like a savings account that cheap calls pay into (by increasing the potential) to compensate for later expensive calls (which decrease the potential). In a nutshell: the less “balanced” a data structure is, the higher its potential should be because it will be needed to pay for the impending restructuring.

The **amortized running time** (or complexity) is defined as the actual running time plus the difference in potential, i.e. the potential after the call minus the potential before it. If the potential increases, the amortized running time is higher than the actual running time and we pay the difference into our savings account. If the potential decreases, the amortized running time is lower than the actual running time and we take something out of our savings account to pay for the difference.

More formally, we are given some data structure with operations *f*, *g*, etc., with corresponding time functions T_f, T_g, etc. We are also given a potential function Φ. The amortized running time function A_f for f is defined as follows:

$$A_f\ s = T_f\ s + \Phi\ (f\ s) - \Phi\ s \tag{19.1}$$

where s is the data structure under consideration; f may also have additional parameters. Given a sequence of data structure states $s_0, \ldots, s_n$ where $s_{i+1} = f\ s_i$, it is not hard to see that

$$\sum_{i=0}^{n-1} A_f\ s_i = \sum_{i=0}^{n-1} T_f\ s_i + \Phi\ s_n - \Phi\ s_0$$

If we assume (for simplicity) that $\Phi\ s_0 = 0$, then it follows immediately that the amortized running time of the whole sequence is an upper bound of the actual running time (because Φ is non-negative). This observation becomes useful if we can bound $A_f\ s$ by some closed term $ub_f\ s$. Typical examples for $ub_f\ s$ are constants, logarithms, etc. Then we can conclude that f has constant, logarithmic, etc. amortized complexity. Thus the only proof obligation is

$$A_f\ s \le ub_f\ s$$

possibly under the additional assumption *invar s* if the data structure comes with an invariant *invar*.

In the sequel we assume that s_0 is some fixed value, typically “empty”, and that its potential is 0.

How do we analyze operations that combine two data structures, e.g. the union of two sets? Their amortized complexity can be defined in analogy to (19.1):

$$A_g\ s_1\ s_2 = T_g\ s_1\ s_2 + \Phi\ (g\ s_1\ s_2) - (\Phi\ s_1 + \Phi\ s_2)$$

So far we implicitly assumed that all operations return the data structure as a result, otherwise $\Phi\ (f\ s)$ does not make sense. How should we analyze so-called **observer functions** that do not modify the data structure but return a value of some other type? Because the data structure is not modified, we have $s_{i+1} = s_i$ and thus $A_f\ s = T_f\ s$. In a nutshell, amortized analysis is irrelevant for pure observer functions.

Now we study two classical examples of amortize analyses. More complex applications are found in later chapters.

19.2 Binary Counter

The binary counter is represented by a list of Booleans where the head of the list is the least significant bit. The increment operation and its running time are easily defined:

$$
\begin{array}{l}
\mathit{incr} :: \mathit{bool\ list} \Rightarrow \mathit{bool\ list} \\
\mathit{incr}\ [] = [\mathit{True}] \\
\mathit{incr}\ (\mathit{False}\ \#\ bs) = \mathit{True}\ \#\ bs \\
\mathit{incr}\ (\mathit{True}\ \#\ bs) = \mathit{False}\ \#\ \mathit{incr}\ bs \\
\\
T_{\mathit{incr}} :: \mathit{bool\ list} \Rightarrow \mathit{real} \\
T_{\mathit{incr}}\ [] = 1 \\
T_{\mathit{incr}}\ (\mathit{False}\ \#\ \_) = 1 \\
T_{\mathit{incr}}\ (\mathit{True}\ \#\ bs) = T_{\mathit{incr}}\ bs + 1
\end{array}
$$

The potential of a counter is the number of *True*'s because they increase T_{incr}:

$$
\begin{array}{l}
\Phi :: \mathit{bool\ list} \Rightarrow \mathit{real} \\
\Phi\ bs = |\mathit{filter}\ (\lambda x.\ x)\ bs|
\end{array}
$$

Clearly the potential is never negative.

The amortized complexity of *incr* is 2:

$$T_{\mathit{incr}}\ bs + \Phi\ (\mathit{incr}\ bs) - \Phi\ bs = 2$$

This can be proved automatically by induction on bs.

19.3 Dynamic Tables

A **dynamic table** is an abstraction of a dynamic array that can grow and shrink subject to a specific memory management. At any point the table has a certain **size** (= number of cells) but some cells may be free. As long as there are free cells, inserting a new element into the table takes constant time. When the table overflows, the whole table has to be copied into a larger table, which takes linear time. Similarly, elements can be deleted from the table in constant time, but when too many elements have been deleted, the table is contracted to save space. Contraction involves copying into a smaller table. This is an abstraction of a dynamic array, where the index bounds can grow and shrink. It is an abstraction because we ignore the actual contents of the table and abstract the table to a pair $(n,\ l)$ where l is its size and $n < l$ the number of occupied cells. The empty table is represented by $(0,\ 0)$.

Below we state the complexity results only informally, e.g. "The amortized cost of insertion is 3", because the formal counterpart $A_{ins}\ s = 3$ is obvious. Nor do we comment on the formal proofs because they are essentially just case analyses (as dictated by the definitions) plus (linear) arithmetic.

19.3.1 Insertion

The key observation is that doubling the size of the table upon overflow leads to an amortized cost of 3 per insertion: 1 for inserting the element, plus 2 towards the later cost of copying a table of size l upon overflow (because only the $l/2$ elements that lead to the overflow pay for it).

Insertion always increments n by 1. The size increases from 0 to 1 with the first insertion and doubles with every further overflow:

```
ins :: nat × nat ⇒ nat × nat
ins (n, l) = (n + 1, if n < l then l else if l = 0 then 1 else 2 · l)
```

This guarantees the **load factor** n/l is always between 1/2 and 1:

```
invar :: nat × nat ⇒ bool
invar (n, l) = (l/2 ≤ n ∧ n ≤ l)
```

Function T_{ins} below is not derived from *ins* (otherwise it would be 0), but from a version of *ins* that acts on an actual table and performs copying upon overflow:

```
T_ins :: nat × nat ⇒ real
T_ins (n, l) = (if n < l then 1 else if l = 0 then 1 else n + 1)
```

The potential of a table (n, l) is $2 \cdot (n - l/2) = 2 \cdot n - l$ following the intuitive argument at the beginning of the Insertion section.

```
Φ :: nat × nat ⇒ real
Φ (n, l) = 2 · n − l
```

The potential is always non-negative because of the invariant.

Note that in our informal explanatory text we use "/" freely and assume we are working with real numbers. In the formalization we often prefer multiplication over division because the former is easier to reason about.

19.3.2 Insertion and Deletion

A naive implementation of deletion simply removes the element but never contracts the table. This works (Exercise 19.2) but wastes space. It is tempting to think we should contract once the load factor drops below 1/2. However, this can lead to fluttering: Starting with a full table of size l, one insertion causes overflow, two deletions cause contraction, two insertion causes overflow, and so on. The cost of each overflow and contraction is l but there are at most two operations to pay for it. Thus the amortized cost of both insertion and deletion cannot be constant. It turns out that it works if we allow the load factor to drop to 1/4 before we contract the table to half its size:

$$\textit{del} :: nat \times nat \Rightarrow nat \times nat$$
$$\textit{del}\ (n,\ l) = (n - 1,\ \textbf{if}\ n = 1\ \textbf{then}\ 0\ \textbf{else if}\ 4 \cdot (n - 1) < l\ \textbf{then}\ l\ \mathrm{div}\ 2\ \textbf{else}\ l)$$

$$T_{del} :: nat \times nat \Rightarrow real$$
$$T_{del}\ (n,\ l) = (\textbf{if}\ n = 1\ \textbf{then}\ 1\ \textbf{else if}\ 4 \cdot (n - 1) < l\ \textbf{then}\ n\ \textbf{else}\ 1)$$

Now the load factor is always between 1/4 and 1. It turns out that the lower bound is not needed in the proofs and we settle for a simpler invariant:

$$\textit{invar} :: nat \times nat \Rightarrow bool$$
$$\textit{invar}\ (n,\ l) = (n \le l)$$

The potential distinguishes two cases:

$$\Phi :: nat \times nat \Rightarrow real$$
$$\Phi\ (n,\ l) = (\textbf{if}\ n < l/2\ \textbf{then}\ l/2 - n\ \textbf{else}\ 2 \cdot n - l)$$

The condition $2 \cdot n \ge l$ concerns the case when we are heading up for an overflow and has been dealt with above. Conversely, $2 \cdot n < l$ concerns the case where we are heading down for a contraction. That is, we start at $(l,\ 2 \cdot l)$ (where the potential is 0) and $l/2$ deletions lead to $(l/2,\ 2 \cdot l)$ where a contraction requires $l/2$ credits, and indeed $\Phi\ (l/2,\ 2 \cdot l) = l/2$. Since $l/2$ is spread over $l/2$ deletions, the amortized cost of a single deletion is 2, 1 for the real cost and 1 for the savings account. The amortized cost of insertion is unchanged.

Note that the case distinction in the definition of Φ ensures that the potential is always ≥ 0 — the invariant is not even needed.

19.4 Exercises

Exercise 19.1. Generalize the binary counter to a base b counter, $b \geq 2$. Prove that there is a constant c such that the amortized complexity of incrementation is at most c for every $b \geq 2$.

Exercise 19.2. Prove that in the dynamic table with naive deletion (where deletion decrements n but leaves l unchanged), insertion has an amortized cost of at most 3 and deletion of at most 1.

Exercise 19.3. Modify deletion as follows. Contraction happens when the load factor would drop below 1/3, i.e. when $3 \cdot (n - 1) < l$. Then the size of the table is multiplied by 2/3, i.e. reduced to $(2 \cdot l)$ div 3. Prove that insertion and deletion have constant amortized complexity using the potential $\Phi\ (n,\ l) = |2 \cdot n - l|$.

Chapter Notes

Amortized analysis is due to Tarjan [1985]. Introductions to it can be found in most algorithm textbooks. This chapter is based on work by Nipkow [2015] and Nipkow and Brinkop [2019] which also formalizes the meta-theory of amortized analysis.

20 Queues

Alejandro Gómez-Londoño and Tobias Nipkow

20.1 Queue Specification

A **queue** can be viewed as a glorified list with function *enq* for adding an element to the end of the list and function *first* for accessing and *deq* for removing the first element. This is the full ADT:

ADT Queue =

interface $\mathit{empty} :: {}'q$
$\mathit{enq} :: {}'a \Rightarrow {}'q \Rightarrow {}'q$
$\mathit{deq} :: {}'q \Rightarrow {}'q$
$\mathit{first} :: {}'q \Rightarrow {}'a$
$\mathit{is\_empty} :: {}'q \Rightarrow \mathit{bool}$

abstraction $\mathit{list} :: {}'q \Rightarrow {}'a\ \mathit{list}$

invariant $\mathit{invar} :: {}'q \Rightarrow \mathit{bool}$

specification $\mathit{list}\ \mathit{empty} = []$

$$
\begin{array}{l}
\mathit{invar}\ q \longrightarrow \mathit{list}\ (\mathit{enq}\ x\ q) = \mathit{list}\ q\ @\ [x] \\
\mathit{invar}\ q \longrightarrow \mathit{list}\ (\mathit{deq}\ q) = \mathsf{tl}\ (\mathit{list}\ q) \\
\mathit{invar}\ q \land \mathit{list}\ q \neq [] \longrightarrow \mathit{first}\ q = \mathsf{hd}\ (\mathit{list}\ q) \\
\mathit{invar}\ q \longrightarrow \mathit{is\_empty}\ q = (\mathit{list}\ q = []) \\
\\
\mathit{invar}\ \mathit{empty} \\
\mathit{invar}\ q \longrightarrow \mathit{invar}\ (\mathit{enq}\ x\ q) \\
\mathit{invar}\ q \longrightarrow \mathit{invar}\ (\mathit{deq}\ q)
\end{array}
$$

A trivial implementation is as a list, but then *enq* is linear in the length of the queue. To improve this we consider two more sophisticated implementations. First, a simple implementation where every operation has amortized constant complexity. Second, a tricky "real time" implementation where every operation has worst-case constant complexity.

20.2 Queues as Pairs of Lists

The queue is implemented as a pair of lists $(fs,\ rs)$, the front and rear lists. Function *enq* adds elements to the head of the rear rs and *deq* removes elements from the head

```
norm :: 'a list × 'a list ⇒ 'a list × 'a list
norm (fs, rs) = (if fs = [] then (itrev rs [], []) else (fs, rs))

enq :: 'a ⇒ 'a list × 'a list ⇒ 'a list × 'a list
enq a (fs, rs) = norm (fs, a # rs)

deq :: 'a list × 'a list ⇒ 'a list × 'a list
deq (fs, rs) = (if fs = [] then (fs, rs) else norm (tl fs, rs))

first :: 'a list × 'a list ⇒ 'a
first (a # _, _) = a

is_empty :: 'a list × 'a list ⇒ bool
is_empty (fs, _) = (fs = [])
```

Figure 20.1 Queue as a pair of lists

of the front fs. When fs becomes empty, it is replaced by *rev* rs (and rs is emptied) — the reversal ensures that now the oldest element is at the head. Hence rs is really the reversal of the rear of the queue but we just call it the rear. The abstraction function is obvious:

```
list :: 'a list × 'a list ⇒ 'a list
list (fs, rs) = fs @ rev rs
```

Clearly *enq* and *deq* are constant-time until the front becomes empty. Then we need to reverse the rear which takes linear time (if it is implemented by *itrev*, see Section 1.5.1). But we can pay for this linear cost up front by paying a constant amount for each call of *enq*. Thus we arrive at amortized constant time. See below for the formal treatment.

The implementation is shown in Figure 20.1. Of course *empty* = ([], []). Function *norm* performs the reversal of the rear once the front becomes empty. Why does not only *deq* but also *enq* call *norm*? Because otherwise *enq* x_n (...(*enq* x_1 *empty*)...) would result in ([], [x_n, ..., x_1]) and *first* would become an expensive operation because it would require the reversal of the rear. Thus we need to avoid queues $([], rs)$ where $rs \neq []$. Thus *norm* guarantees the following invariant:

$$\mathit{invar} :: {}'a\ \mathit{list} \times {}'a\ \mathit{list} \Rightarrow \mathit{bool}$$
$$\mathit{invar}\ (\mathit{fs}, \mathit{rs}) = (\mathit{fs} = [] \longrightarrow \mathit{rs} = [])$$

Functional correctness, i.e. proofs of the properties in the ADT *Queue*, are straightforward. Let us now turn to the amortized running time analysis. The time functions are shown in Appendix B.8.

For the amortized analysis we define the potential function

$$\Phi :: {}'a\ \mathit{list} \times {}'a\ \mathit{list} \Rightarrow \mathit{nat}$$
$$\Phi\ (\_, \mathit{rs}) = |\mathit{rs}|$$

because $|rs|$ is the amount we have accumulated by charging 1 for each *enq*. This is enough to pay for the eventual reversal. Now it is easy to prove that both *enq* and *deq* have amortized constant running time:

$$T_{enq}\ a\ (\mathit{fs}, \mathit{rs}) + \Phi\ (\mathit{enq}\ a\ (\mathit{fs}, \mathit{rs})) - \Phi\ (\mathit{fs}, \mathit{rs}) \leq 2$$
$$T_{deq}\ (\mathit{fs}, \mathit{rs}) + \Phi\ (\mathit{deq}\ (\mathit{fs}, \mathit{rs})) - \Phi\ (\mathit{fs}, \mathit{rs}) \leq 1$$

The two observer functions *first* and *is_empty* have constant running time.

Exercise 20.1. A **min-queue** is a queue that supports an operation *minq* that returns the minimal element in the queue. Formally, the ADT *Queue* is extended as follows: we assume ${}'a :: \mathit{linorder}$, extend the interface with $\mathit{minq} :: {}'q \Rightarrow {}'a$ and the specification with

$$\mathit{invar}\ q \wedge \mathit{list}\ q \neq [] \longrightarrow \mathit{minq}\ q = \mathit{Min}\ (\mathit{set}\ (\mathit{list}\ q))$$

Implement and verify a min-queue with amortized constant time operations. Hint: follow the pair-of-lists idea above but store additional information that allows you to return the minimal element in constant time.

20.3 A Real-Time Implementation

This sections presents the **Hood-Melville queue**, a tricky implementation that improves upon the representation in the previous section by preemptively performing reversals over a number of operations before they are required.

20.3.1 Stepped Reversal

Breaking down a reversal operation into multiple steps can be done using the following function:

```
rev_step :: 'a list × 'a list ⇒ 'a list × 'a list
rev_step (x # xs, ys) = (xs, x # ys)
rev_step ([], ys) = ([], ys)
```

where $x \ \# \ xs$ is the list being reversed, and $x \ \# \ ys$ is the partial reversal result. Thus, to reverse a list of size 3 one should call *rev_step* 3 times:

$$\begin{aligned}
&\mathit{rev\_step}\ ([1,\ 2,\ 3],\ [])=([2,\ 3],\ [1])\\
&\mathit{rev\_step}\ (\mathit{rev\_step}\ ([1,\ 2,\ 3],\ []))=([3],\ [2,\ 1])\\
&\mathit{rev\_step}\ (\mathit{rev\_step}\ (\mathit{rev\_step}\ ([1,\ 2,\ 3],\ [])))=([],\ [3,\ 2,\ 1])
\end{aligned}$$

Note that each call to *rev_step* takes constant time since its definition is non-recursive.

Using the notation f^n for the n-fold composition of function f we can state a simple inductive lemma:

Lemma 20.1. $\mathit{rev\_step}^{|xs|}\ (xs,\ ys) = ([],\ \mathit{rev}\ xs \mathbin{@} ys)$

As a special case this implies $\mathit{rev\_step}^{|xs|}\ (xs,\ []) = ([],\ \mathit{rev}\ xs)$.

20.3.2 A Real-Time Intuition

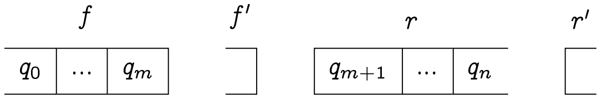
Hood-Melville queues are similar to those presented in Section 20.2 in that they use a pair of lists $(f,\ r)$ (front and rear — for succinctness we drop the s's now) to achieve constant running time *deq* and *enq*. However, they avoid a costly reversal operation once f becomes empty by preemptively computing a new front $\mathit{fr} = f \mathbin{@} \mathit{rev}\ r$ one step at a time using *rev_step* as enqueueing and dequeueing operations occur. The process that generates fr consists of three phases:

1. Reverse r to form r', which is the tail end of fr
2. Reverse f to form f'
3. Reverse f' onto r' to form fr

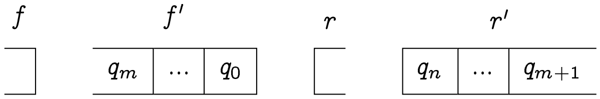
All three phases can be described in terms of *rev_step* as follows:

1. $r' = \mathit{snd}\ (\mathit{rev\_step}^{|r|}\ (r,\ []))$
2. $f' = \mathit{snd}\ (\mathit{rev\_step}^{|f|}\ (f,\ []))$
3. $\mathit{fr} = \mathit{snd}\ (\mathit{rev\_step}^{|f'|}\ (f',\ r'))$

Phases (1) and (2) are independent and can be performed at the same time. Hence, when starting from this configuration,



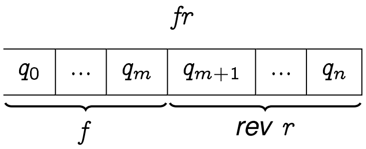
after *max* $|f|$ $|r|$ steps of reversal, the state would be the following:



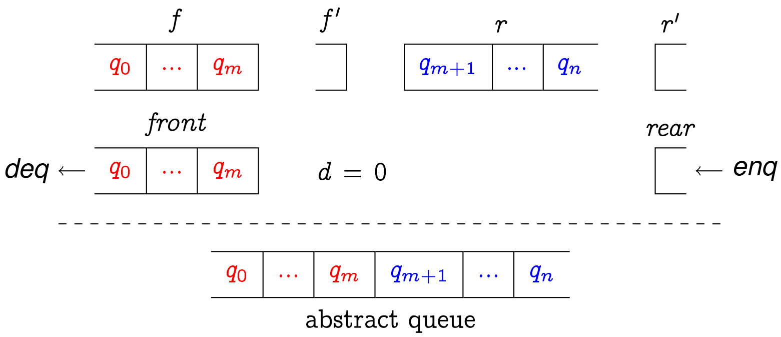
Phase (3) reverses f' onto r' to obtain the same result as a call to *list*:

$$
\begin{aligned}
fr &= \mathit{snd}\ (\mathit{rev\_step}^{|f'|}\ (f',\ r')) && \text{by definition of } fr \\
&= \mathit{rev}\ f'\ @\ r' && \text{using Lemma 20.1} \\
&= \mathit{rev}\ f'\ @\ \mathit{snd}\ (\mathit{rev\_step}^{|r|}\ (r,\ [])) && \text{by definition of } r' \\
&= \mathit{rev}\ f'\ @\ \mathit{rev}\ r && \text{using Lemma 20.1} \\
&= \mathit{rev}\ (\mathit{snd}\ (\mathit{rev\_step}^{|f|}\ (f,\ [])))\ @\ \mathit{rev}\ r' && \text{by definition of } f' \\
&= \mathit{rev}\ (\mathit{rev}\ f)\ @\ \mathit{rev}\ r && \text{using Lemma 20.1} \\
&= f\ @\ \mathit{rev}\ r && \text{by } \mathit{rev} \text{ involution}
\end{aligned}
$$

The resulting front list *fr* contains all elements previously in f and r:



A Hood-Melville queue spreads all reversal steps across queue-altering operations, requiring careful bookkeeping. To achieve this gradual reversal, additional lists *front* and *rear* are used for enqueuing and dequeuing, while internal operations rely only on f, f', r, and r'. At the start of the reversal process, *rear* is copied into r and emptied; similarly, *front* is copied into f, but its contents are kept as they might need to be dequeued. Moreover, to avoid using elements from f or f' that may have been removed from *front*, a counter d records the number of dequeuing operations that have occurred since the reversal process started; this way, only $|f'| - d$ elements are appended into r to form *fr*. Once the reversal finishes, *fr* become the new *front* and the internal lists are cleared. When the queue is not being reversed, all operations are performed in a manner similar to previous implementations. The configuration of a queue at the beginning of the reversal process is as follows:



20.3.3 The Reversal Strategy

A crucial detail of this implementation is determining at which point the reversal process should start. The strategy is to start once $|\mathit{rear}|$ becomes larger than $|\mathit{front}|$. This ensures that all reversal steps are done before *front* runs out of elements or *rear* becomes larger than the new front (*fr*).

With this strategy, once $|\mathit{rear}| = n+1$ and $|\mathit{front}| = n$, the reversal processes starts. The first two phases take $n+1$ steps ($\mathsf{max}\ |\mathit{front}|\ |\mathit{rear}|$) to generate f' and r', and the third phase produces *fr* in n steps. A complete reversal takes $2n+1$ steps. Because the queue can only perform n *deq* operations before *front* is exhausted, $2n+1$ steps must be performed in at most n operations. This can be achieved by performing the first two steps in the operation that causes *rear* to become larger than *front* and two more steps in each subsequent operation. Therefore, $2(n+1)$ steps can occur before *front* is emptied, allowing the reversal process to finish in time.

Finally, since at most n *enq* or *deq* operations can occur during reversal, the largest possible *rear* has length n (only *enq* ops), while the smallest possible *fr* has length $n+1$ (only *deq* ops). Thus, after the reversing process has finished, the new front (*fr*) is always larger than *rear*.

20.3.4 Implementation

Queues are implemented using the following record type:

```
record 'a queue = lenf :: nat
                  front :: 'a list
                  status :: 'a status
                  rear :: 'a list
                  lenr :: nat
```

A record is a product type with named fields and inbuilt construction, selection, and update operations. Values of $'a\ queue$ are constructed using $\mathsf{make} :: nat \Rightarrow {'a}\ list \Rightarrow {'a}\ status \Rightarrow {'a}\ list \Rightarrow nat \Rightarrow {'a}\ queue$ were each argument corresponds to one of the fields of the record in canonical order. Additionally, given a queue q we can obtain the value of, for example, field *front* with *front* q, and update its content using $q(\!|\mathit{front} := []|\!)$. Multiple updates can be composed, e.g. $q(\!|\mathit{front} := [],\ \mathit{rear} := []|\!)$.

All values in the queue, along with its internal state, are stored in the various fields of $'a\ queue$. Fields *front* and *rear* contain the lists over which all queue operations are performed. The lengths of *front* and *rear* are recorded in *lenf* and *lenr* to avoid calling *length*, whose complexity is not constant. Finally, *status* tracks the current reversal phase of the queue in a $'a\ status$ value:

```
datatype 'a status =
    Idle |
    Rev nat ('a list) ('a list) ('a list) ('a list) |
    App nat ('a list) ('a list) |
    Done ('a list)
```

Each value of $'a\ status$ represents either a phase of reversal or the queue's normal operation. Constructor *Idle* signals that no reversal is being performed. Status *Rev* $ok\ f\ f'\ r\ r'$ corresponds to phases (1) and (2) where the lists f, f', r, and r' are used for the reversal steps of the front and the rear. The *App* $ok\ f'\ r'$ case corresponds to phase (3) where both lists are appended to form the new front (fr). In both *App* and *Rev*, the first argument $ok :: nat$ keeps track of the number of elements in f' that have not been removed from the queue, effectively $ok = |f'| - d$, where d is the number of *deq* operations that have occurred so far. Last, *Done* fr marks the end of the reversal process and contains only the new front list fr.

In the implementation, all of the steps of reversal operations in the queue are performed by functions *exec* and *invalidate*; they ensure at each step that the front list being computed is kept consistent w.r.t. the contents and operations in the queue.

Function $\mathsf{exec} :: {'a}\ status \Rightarrow {'a}\ status$ performs the incremental reversal of the front list by altering the queue's *status* one step at a time in accordance with the reversal phases. Following the strategy described in Section 20.3.3, all queue operations call *exec* twice to be able to finish the reversal in time. On *Idle* queues *exec* has no effect. The implementation of *exec* is an extension of *rev_step* with specific considerations for each *status* value and is defined as follows:

```
exec :: 'a status ⇒ 'a status
exec (Rev ok (x # f) f' (y # r) r') = Rev (ok + 1) f (x # f') r (y # r')
exec (Rev ok [] f' [y] r') = App ok f' (y # r')
exec (App 0 _ r') = Done r'
exec (App ok (x # f') r') = App (ok − 1) f' (x # r')
exec s = s
```

If the *status* is *Rev* $ok\, f\, f'\ r\, r'$, then *exec* performs two (or one if $f = []$) simultaneous reversal steps from f and r into f' and r'; moreover ok is incremented if a new element has been added to f'. Once f is exhausted and r is a singleton list, the remaining element is moved into r' and the *status* is updated to the next phase of reversal. In the *App* $ok\ f'\ r'$ phase, *exec* moves elements from f' to r' until $ok = 0$, at which point r' becomes the new front by transitioning to *Done* r'. In all other cases *exec* behaves like the identity function. As is apparent from its implementation, a number of assumptions are required for *exec* to function properly and eventually produce *Done*. These assumption are discussed in Section 20.3.5.

If an element is dequeued during the reversal process, it also needs to be removed from the new front list (fr) being computed. Function *invalidate* does this:

```
invalidate :: 'a status ⇒ 'a status
invalidate (Rev ok f f' r r') = Rev (ok − 1) f f' r r'
invalidate (App 0 _ (_ # r')) = Done r'
invalidate (App ok f' r') = App (ok − 1) f' r'
invalidate s = s
```

By decreasing the value of ok, the number of elements from f' that are moved into r' in phase (3) is reduced; since *exec* produces *Done* early, once $ok = 0$, the remaining elements of f' are ignored. Furthermore, since f' is a reversal of the front list, elements left behind in its tail correspond directly to those being removed from the queue.

The rest of the implementation is shown below. Auxiliary function *exec2* applies *exec* twice and updates the queue accordingly if *Done* is returned.

```
exec2 :: 'a queue ⇒ 'a queue
exec2 q = (case exec (exec (status q)) of
             Done fr ⇒ q⦇status := Idle, front := fr⦈ |
             st ⇒ q⦇status := st⦈)
```

```
check :: 'a queue ⇒ 'a queue
check q
= (if lenr q ≤ lenf q then exec2 q
   else exec2
          (q⦇lenf := lenf q + lenr q, status := Rev 0 (front q) [] (rear q) [],
              rear := [], lenr := 0⦈))

empty :: 'a queue
empty = make 0 [] Idle [] 0

first :: 'a queue ⇒ 'a
first q = hd (front q)

enq :: 'a ⇒ 'a queue ⇒ 'a queue
enq x q = check (q⦇rear := x # rear q, lenr := lenr q + 1⦈)

deq :: 'a queue ⇒ 'a queue
deq q
= check
   (q⦇lenf := lenf q − 1, front := tl (front q), status := invalidate (status q)⦈)
```

The two main queue operations, *enq* and *deq*, alter *front* and *rear* as expected and update *lenf* and *lenr* accordingly. To perform all "internal" operations, both functions call *check*. Additionally, *deq* uses *invalidate* to mark elements as removed.

Function *check* calls *exec2* if *lenr* is not larger than *lenf*. Otherwise a reversal process is initiated: *rear* is emptied and *lenr* is set to 0; *lenf* is increased to the size of the whole queue since, conceptually, all element are now in the soon-to-be-computed front; the new status is initialized as described in Section 20.3.2.

The time complexity of this implementation is clearly constant, since there are no recursive functions.

20.3.5 Correctness

To show this implementation is an instance of the ADT *Queue*, we need a number of invariants to ensure the consistency of *'a queue* values are preserved by all operations.

Initially, as hinted by the definition of *exec*, values of type *'a status* should have specific properties to guarantee a *Done* result after a small enough number of calls to *exec*. The predicate *inv_st* defines these properties as follows:

```
inv_st :: 'a status ⇒ bool
inv_st (Rev ok f f' r r') = (|f| + 1 = |r| ∧ |f'| = |r'| ∧ ok ≤ |f'|)
inv_st (App ok f' r') = (ok ≤ |f'| ∧ |f'| < |r'|)
inv_st Idle = True
inv_st (Done _) = True
```

Case *Rev* $ok\ f\ f'\ r\ r'$ ensures that f and r follow the reversal strategy, and counter ok is only ever increased as elements are added to f'. Similarly, for *App* $ok\ f'\ r'$, it must follow that r' remains larger than f', and $|f'|$ provides an upper bound for ok.

The *queue* invariant *invar* is an extension of *inv_st* and considers all the other fields in the queue:

```
invar :: 'a queue ⇒ bool
invar q
= (lenf q = |front_list q| ∧ lenr q = |rear_list q| ∧ lenr q ≤ lenf q ∧
    (case status q of
     Rev ok f f' _ _ ⇒
       2 · lenr q ≤ |f'| ∧ ok ≠ 0 ∧ 2 · |f| + ok + 2 ≤ 2 · |front q|
    | App ok _ r ⇒ 2 · lenr q ≤ |r| ∧ ok + 1 ≤ 2 · |front q|
    | _ ⇒ True) ∧
    (∃rest. front_list q = front q @ rest) ∧
    (∄fr. status q = Done fr) ∧
    inv_st (status q))
```

The condition *lenr* $q = |$*rear_list* $q|$ ensures *lenr* is equal to the length of the queue's rear, where function *rear_list* = *rev* ∘ *rear* produces the rear list in canonical order. Similarly for *lenf* $q = |$*front_list* $q|$ where *front_list* warrants special attention because it must compute the list representing the front of the queue even during a reversal:

```
front_list :: 'a queue ⇒ 'a list
front_list q = (case status q of
                 Idle ⇒ front q |
                 Rev ok f f' r r' ⇒ rev (take ok f') @ f @ rev r @ r' |
                 App ok f' x ⇒ rev (take ok f') @ x |
                 Done f ⇒ f)
```

In case $\mathit{App}\ ok\ f'\ r'$, the front list corresponds to the final result of the stepped reversal (20.1), but only elements in f' that are still in the queue, denoted by $\mathit{take}\ ok\ f'$, are considered. Analogously for $\mathit{Rev}\ ok\ f\ f'\ r\ r'$, both stepped reversal results are appended and only relevant elements in f' are used, however, rear lists r and r' are reversed again to achieve canonical order.

Continuing with *invar*, inequality $\mathit{lenr}\ q \leq \mathit{lenf}\ q$ is the main invariant in our reversal strategy, and by the previous two equalities must hold even as internal operations occur. Furthermore, $\exists rest.\ \mathit{front\_list}\ q = \mathit{front}\ q\ @\ rest$ ensures $\mathit{front}\ q$ is contained within $\mathit{front\_list}\ q$, thus preventing any mismatch between the internal state and the queue's front. Given that *exec2* is the only function that manipulates a queue's *status*, it holds that $\nexists fr.\ \mathit{status}\ q = \mathit{Done}\ fr$ since any internal *Done* result is replaced by *Idle*.

The case distinction on $\mathit{status}\ q$ places size bounds on internal lists *front* and *rear* ensuring the front does not run out of elements and the rear never grows beyond $\mathit{lenr}\ q \leq \mathit{lenf}\ q$. In order to clarify this part of *invar*, consider the following correspondences, which hold once the reversal process starts:

- $\mathit{lenr}\ q$ corresponds to the number of *enq* operations performed so far, and $2 \cdot \mathit{lenr}\ q$ denotes the number of *exec* applications in those operations.
- $|\mathit{front}\ q|$ corresponds to the number of *deq* operations that can be performed before $\mathit{front}\ q$ is exhausted. Therefore, $2 \cdot |\mathit{front}\ q|$ is the minimum number of *exec* applications the queue must perform to complete the reversal in time.
- In case $\mathit{Rev}\ ok\ f\ f'\ r\ r'$, $|f'|$ corresponds to the number of *exec*'s performed so far and the internal length of front being constructed. Expression $|r|$ is the analogue for $\mathit{App}\ ok\ f\ r$.
- From a well formed $\mathit{App}\ ok\ f\ r$ it takes $ok + 1$ applications of *exec* to reach *Done*: the base case of *App* is reached after ok applications, and the transition to *Done* takes an extra step.
- From a well formed $\mathit{Rev}\ ok\ f\ f'\ r\ r'$ it takes $2 \cdot |f'| + ok + 2$ applications of *exec* to reach *Done*: the base case of *Rev* is reached after $|f'|$ applications (incrementing ok by the same amount), the transition to *App* takes one step, and $ok + |f'|$ extra steps are needed to reach *Done* from *App*.

In the $\mathit{Rev}\ ok\ f\ f'\ r\ r'$ case, $2 \cdot \mathit{lenr}\ q \leq |f'|$ ensures f' grows larger with every *enq* operation and the internal list is at least twice the length of the queue's rear. Additionally, the value of ok cannot be 0 as this either marks the beginning of a reversal which calls *exec2* immediately, or signals that elements in $\mathit{front}\ q$ have run out. Finally, to guarantee the reversal process can finish before the $\mathit{front}\ q$ is exhausted the number of *exec* applications before reaching *Done* must be less than the minimum number of applications required, denoted by $2 \cdot |f| + ok + 2 \leq 2 \cdot |\textit{front}\ q|$.

Case *App* $ok\ f\ r$ has similar invariants, with equation $2 \cdot \textit{lenr}\ q \leq |r|$ bounding the growth of r as it was previously done with f'. Moreover, $ok + 1 \leq 2 \cdot |\textit{front}\ q|$ ensures *front* q is not exhausted before the reversal is completed.

With the help of *invar* and this abstraction function

$$\textit{list} :: {'a}\ queue \Rightarrow {'a}\ list$$
$$\textit{list}\ q = \textit{front\_list}\ q \mathbin{@} \textit{rear\_list}\ q$$

all properties of the *Queue* ADT can be proved. The proofs are mostly by cases on the *status* field followed by reasoning about lists. It is essential that the invariant characterizes all cases precisely.

Chapter Notes

The representation of queues as pairs of lists is due to Burton [1982]. Hood-Melville queues are due to Hood and Melville [1981]. The implementation is based on the presentation by Okasaki [1998, section 8.2.1.].

The idea underlying Hood-Melville queues can be generalized to **double-ended queues**. This was explained by Hood [1982, section 4.2], rediscovered in more detail by Chuang and Goldberg [1993] and verified by Tóth and Nipkow [2023].

Okasaki [1998] shows how both single and double-ended real-time queues can be defined more simply with the help of lazy evaluation. However, reasoning about the running time under lazy evaluation is nontrivial, as the verification by Pottier et al. [2024] of the amortized running time of some queue implementations shows.

21 Splay Trees

Tobias Nipkow

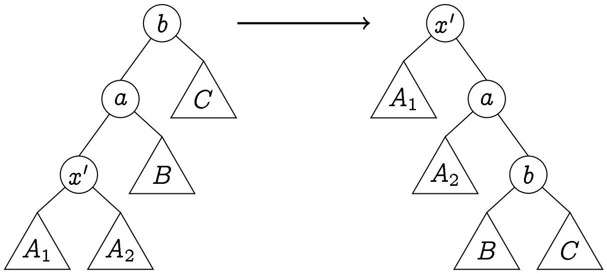
Splay trees are fascinating self-organizing search trees: the tree is modified upon access (including *isin*) to improve the performance of subsequent operations. Concretely, every splay tree operation moves the element concerned to the root. Thus splay trees excel in applications where a small fraction of the entries are the targets of most of the operations. In general, splay trees perform as well as any static binary search tree.

Splay trees have two drawbacks. First, their performance guarantees (logarithmic running time of each operation) are only amortized. Self-organizing does not mean self-balancing: splay trees can become unbalanced, in contrast to, for example, red-black trees. Second, because *isin* modifies the tree, splay trees are less convenient to use in a purely functional language.

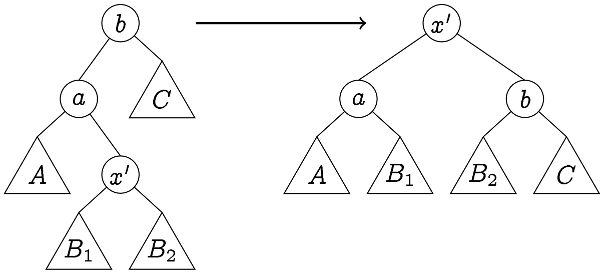
21.1 Implementation

The central operation on splay trees is *splay* :: $'a \Rightarrow 'a\ tree \Rightarrow 'a\ tree$. It searches a tree for a given element x and rotates x (or the last element found before the search for x hits a leaf) to the root by two specific double-rotations (and their mirror images):

zig-zig:



zig-zag:



```
splay x ⟨AB, b, CD⟩
= (case cmp x b of
    LT ⇒ case AB of
           ⟨⟩ ⇒ ⟨AB, b, CD⟩ |
           ⟨A, a, B⟩ ⇒
             case cmp x a of
             LT ⇒ if A = ⟨⟩ then ⟨A, a, ⟨B, b, CD⟩⟩
                   else case splay x A of
                          ⟨A₁, x', A₂⟩ ⇒ ⟨A₁, x', ⟨A₂, a, ⟨B, b, CD⟩⟩⟩ |
             EQ ⇒ ⟨A, a, ⟨B, b, CD⟩⟩ |
             GT ⇒ if B = ⟨⟩ then ⟨A, a, ⟨B, b, CD⟩⟩
                   else case splay x B of
                          ⟨B₁, x', B₂⟩ ⇒ ⟨⟨A, a, B₁⟩, x', ⟨B₂, b, CD⟩⟩ |
    EQ ⇒ ⟨AB, b, CD⟩ |
    GT ⇒ case CD of
           ⟨⟩ ⇒ ⟨AB, b, CD⟩ |
           ⟨C, c, D⟩ ⇒
             case cmp x c of
             LT ⇒ if C = ⟨⟩ then ⟨⟨AB, b, C⟩, c, D⟩
                   else case splay x C of
                          ⟨C₁, x', C₂⟩ ⇒ ⟨⟨AB, b, C₁⟩, x', ⟨C₂, c, D⟩⟩ |
             EQ ⇒ ⟨⟨AB, b, C⟩, c, D⟩ |
             GT ⇒ if D = ⟨⟩ then ⟨⟨AB, b, C⟩, c, D⟩
                   else case splay x D of
                          ⟨D₁, x', D₂⟩ ⇒ ⟨⟨⟨AB, b, C⟩, c, D₁⟩, x', D₂⟩)
```

Figure 21.1 Function *splay*

One of zig-zig and zig-zag is simply the composition of two single rotations (see Section 5.5), one isn't — which one is which?

The full definition of *splay* is shown in Figure 21.1. Function *isin* has a trivial implementation in terms of *splay*:

```
isin :: 'a tree ⇒ 'a ⇒ bool
isin t x = (case splay x t of ⟨⟩ ⇒ False | ⟨_, a, _⟩ ⇒ x = a)
```

Note that *splay* creates a new tree that needs to be returned from a proper *isin* as well, to achieve the amortized logarithmic complexity. This is why splay trees are inconvenient in functional languages. For the moment we ignore this aspect and stick with the above *isin* because it has the type required by the Set ADT.

The implementation of *insert* x t below is straightforward: let $\langle l,\ a,\ r\rangle$ = *splay* x t; if $a = x$, return $\langle l,\ a,\ r\rangle$; otherwise make x the root of a suitable recombination of l, a and r.

```
insert :: 'a ⇒ 'a tree ⇒ 'a tree
insert x t
= (if t = ⟨⟩ then ⟨⟨⟩, x, ⟨⟩⟩
   else case splay x t of
          ⟨l, a, r⟩ ⇒ case cmp x a of
                         LT ⇒ ⟨l, x, ⟨⟨⟩, a, r⟩⟩ |
                         EQ ⇒ ⟨l, a, r⟩ |
                         GT ⇒ ⟨⟨l, a, ⟨⟩⟩, x, r⟩)
```

The implementation of *delete* x t below starts similarly: let $\langle l,\ a,\ r\rangle$ = *splay* x t; if $a \neq x$, return $\langle l,\ a,\ r\rangle$. Otherwise follow the deletion-by-replacing paradigm (Section 5.2.1): if $l \neq \langle\rangle$, splay the maximal element m in l to the root and replace x with it.

```
delete :: 'a ⇒ 'a tree ⇒ 'a tree
delete x t
= (if t = ⟨⟩ then ⟨⟩
   else case splay x t of
          ⟨l, a, r⟩ ⇒
            if x ≠ a then ⟨l, a, r⟩
            else if l = ⟨⟩ then r
                 else case splay_max l of ⟨l', m, _⟩ ⇒ ⟨l', m, r⟩)
```

Function *splay_max* below returns a tree that is just a glorified pair: if $t \neq \langle\rangle$ then *splay_max* t is of the form $\langle t',\ m,\ \langle\rangle\rangle$. The equation *splay_max* $\langle\rangle = \langle\rangle$ is not really needed (*splay_max* is always called with non-$\langle\rangle$ argument) but some lemmas can be stated more simply with this definition.

```
splay_max :: 'a tree ⇒ 'a tree
splay_max ⟨⟩ = ⟨⟩
splay_max ⟨A, a, ⟨⟩⟩ = ⟨A, a, ⟨⟩⟩
splay_max ⟨A, a, ⟨B, b, CD⟩⟩
= (if CD = ⟨⟩ then ⟨⟨A, a, B⟩, b, ⟨⟩⟩
   else case splay_max CD of ⟨C, c, D⟩ ⇒ ⟨⟨⟨A, a, B⟩, b, C⟩, c, D⟩)
```

21.2 Correctness

The *inorder* approach of Section 5.4 applies. Because the details are a bit different (everything is reduced to *splay*) we present the top-level structure.

The following easy inductive properties are used implicitly in a number of subsequent proofs:

$$\textit{splay}\ a\ t = \langle\rangle \longleftrightarrow t = \langle\rangle$$

$$\textit{splay\_max}\ t = \langle\rangle \longleftrightarrow t = \langle\rangle$$

Correctness of *isin*

$$\textit{sorted}\ (\textit{inorder}\ t) \longrightarrow \textit{isin}\ t\ x = (x \in \textit{set}\ (\textit{inorder}\ t))$$

follows directly from this easy inductive property of *splay*:

$$\textit{splay}\ x\ t = \langle l,\ a,\ r\rangle \wedge \textit{sorted}\ (\textit{inorder}\ t) \longrightarrow \\ (x \in \textit{set}\ (\textit{inorder}\ t)) = (x = a)$$

Correctness of *insert* and *delete*

$$\textit{sorted}\ (\textit{inorder}\ t) \longrightarrow \textit{inorder}\ (\textit{insert}\ x\ t) = \textit{ins\_list}\ x\ (\textit{inorder}\ t)$$

$$\textit{sorted}\ (\textit{inorder}\ t) \longrightarrow \textit{inorder}\ (\textit{delete}\ x\ t) = \textit{del\_list}\ x\ (\textit{inorder}\ t)$$

relies on the following characteristic inductive properties of *splay*:

$$\textit{inorder}\ (\textit{splay}\ x\ t) = \textit{inorder}\ t \tag{21.1}$$

$$\textit{sorted}\ (\textit{inorder}\ t) \wedge \textit{splay}\ x\ t = \langle l,\ a,\ r\rangle \longrightarrow \\ \textit{sorted}\ (\textit{inorder}\ l\ @\ x\ \#\ \textit{inorder}\ r)$$

Correctness of *delete* also needs the inductive proposition

$$\textit{splay\_max}\ t = \langle l,\ a,\ r\rangle \wedge \textit{sorted}\ (\textit{inorder}\ t) \longrightarrow \\ \textit{inorder}\ l\ @\ [a] = \textit{inorder}\ t \wedge r = \langle\rangle$$

Note that $\textit{inorder}\ (\textit{splay}\ x\ t) = \textit{inorder}\ t$ is also necessary to justify the proper *isin* that returns the newly created tree as well.

Automation of the above proofs requires the lemmas in Figure 5.2 together with a few additional lemmas about *sorted*, *ins_list* and *del_list* that can be found in the Isabelle proofs.

Recall from Section 5.4 that correctness of *insert* and *delete* implies that they preserve $\mathit{bst} = \mathit{sorted} \circ \mathit{inorder}$. Similarly, (21.1) implies that *splay* preserves *bst*. Thus we may assume the invariant *bst* in the amortized analysis.

These two easy size lemmas are used implicitly below:

$$|\mathit{splay}\ a\ t| = |t| \qquad |\mathit{splay\_max}\ t| = |t|$$

21.3 Amortized Analysis

This section shows that *splay*, insertion and deletion all have amortized logarithmic complexity.

We define the potential Φ of a tree as the sum of the potentials φ of all nodes:

$\Phi :: {}'a\ \mathit{tree} \Rightarrow \mathit{real}$

$\Phi\ \langle\rangle = 0$

$\Phi\ \langle l,\ a,\ r\rangle = \varphi\ \langle l,\ a,\ r\rangle + \Phi\ l + \Phi\ r$

$\varphi\ t \equiv \mathit{lg}\ |t|_1$

The central result is the amortized complexity of *splay*. Function T_{splay} is shown in Appendix B.9. We follow (19.1) and define

$A_{\mathit{splay}}\ a\ t = T_{\mathit{splay}}\ a\ t + \Phi\ (\mathit{splay}\ a\ t) - \Phi\ t$

First we consider the case where the element is in the tree:

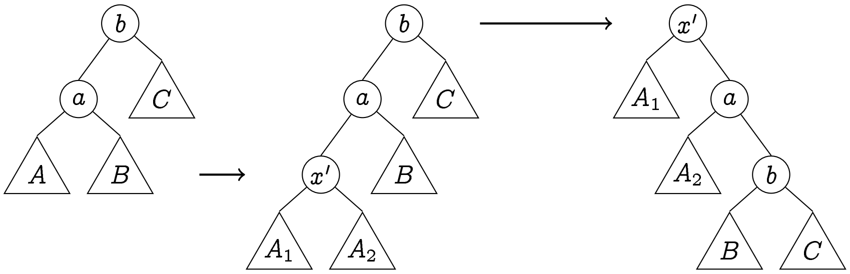
Theorem 21.1. $\mathit{bst}\ t \wedge \langle l,\ x,\ r\rangle \in \mathit{subtrees}\ t \longrightarrow$
$A_{\mathit{splay}}\ x\ t \le 3 \cdot (\varphi\ t - \varphi\ \langle l,\ x,\ r\rangle) + 1$

Proof by induction on the computation of *splay*. The base cases involving $\langle\rangle$ are impossible. For example, consider the call $\mathit{splay}\ x\ t$ where $t = \langle\langle\rangle,\ b,\ C\rangle$ and $x < b$: from $\langle l,\ x,\ r\rangle \in \mathit{subtrees}\ t$ it follows that $x \in \mathit{set\_tree}\ t$ but because $\mathit{bst}\ t$ and $x < b$, this implies that $x \in \mathit{set\_tree}\ \langle\rangle$, a contradiction. There are three feasible base cases. The case $t = \langle\_,\ x,\ \_\rangle$ is easy. We consider one of the two other symmetric cases. Let $t = \langle\langle A,\ x,\ B\rangle,\ b,\ C\rangle$ and $t' = \mathit{splay}\ x\ t = \langle A,\ x,\ \langle B,\ b,\ C\rangle\rangle$.

$$\begin{aligned} A_{\mathit{splay}}\ x\ t &= \Phi\ t' - \Phi\ t + 1 && \text{by definition of } A_{\mathit{splay}} \text{ and } T_{\mathit{splay}} \\ &= \varphi\ t' + \varphi\ \langle B,\ b,\ C\rangle - \varphi\ t - \varphi\ \langle A,\ x,\ B\rangle + 1 && \text{by definition of } \Phi \\ &= \varphi\ \langle B,\ b,\ C\rangle - \varphi\ \langle A,\ x,\ B\rangle + 1 && \text{by definition of } \varphi \end{aligned}$$

$$
\begin{aligned}
&\le \varphi\ t - \varphi\ \langle A,\ x,\ B\rangle + 1 && \text{because } \varphi\ \langle B,\ b,\ C\rangle \le \varphi\ t\\
&\le 3 \cdot (\varphi\ t - \varphi\ \langle A,\ x,\ B\rangle) + 1 && \text{because } \varphi\ \langle A,\ x,\ B\rangle \le \varphi\ t\\
&= 3 \cdot (\varphi\ t - \varphi\ \langle l,\ x,\ r\rangle) + 1 && \text{because } \mathit{bst}\ t \wedge \langle l,\ x,\ r\rangle \in \mathit{subtrees}\ t
\end{aligned}
$$

There are four inductive cases. We consider two of them, the other two are symmetric variants. First the zig-zig case:



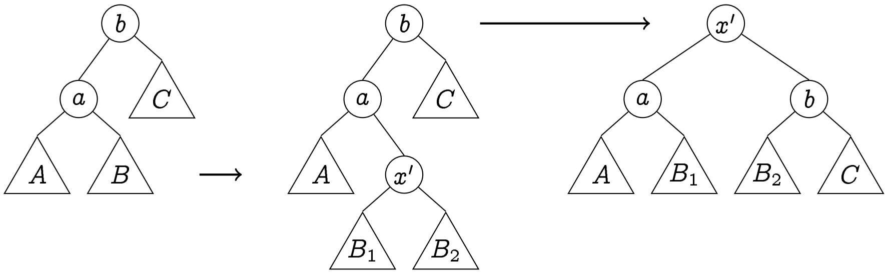
This is the case where $x < a < b$ and $A \ne \langle\rangle$. On the left we have the input and on the right the output of $\mathit{splay}\ x$. Because $A \ne \langle\rangle$, $\mathit{splay}\ x\ A = \langle A_1,\ x',\ A_2\rangle =: A'$ for some A_1, x' and A_2. The intermediate tree is obtained by replacing A by A'. This tree is shown for illustration purpose only; in the algorithm the right tree is constructed directly from the left one. We abbreviate compound trees like $\langle A,\ a,\ B\rangle$ by the names of their subtrees, in this case AB. Similarly $lr = \langle l,\ x,\ r\rangle$. First note that

$$\varphi\ A_1A_2BC = \varphi\ ABC \qquad (*)$$

because $|A'| = |\mathit{splay}\ x\ A| = |A|$. We can now prove the claim:

$$
\begin{aligned}
&A_{\mathit{splay}}\ x\ ABC = T_{\mathit{splay}}\ x\ A + 1 + \Phi\ A_1A_2BC - \Phi\ ABC\\
&= T_{\mathit{splay}}\ x\ A + 1 + \Phi\ A_1 + \Phi\ A_2 + \varphi\ A_2BC + \varphi\ BC - \Phi\ A - \varphi\ AB\\
&&& \text{by } (*) \text{ and definition of } \Phi\\
&= T_{\mathit{splay}}\ x\ A + \Phi\ A' - \varphi\ A' - \Phi\ A + \varphi\ A_2BC + \varphi\ BC - \varphi\ AB + 1\\
&= A_{\mathit{splay}}\ x\ A + \varphi\ A_2BC + \varphi\ BC - \varphi\ AB - \varphi\ A' + 1\\
&\le 3 \cdot \varphi\ A + \varphi\ A_2BC + \varphi\ BC - \varphi\ AB - \varphi\ A' - 3 \cdot \varphi\ lr + 2\\
&&& \text{by IH and } lr \in \mathit{subtrees}\ A\\
&= 2 \cdot \varphi\ A + \varphi\ A_2BC + \varphi\ BC - \varphi\ AB - 3 \cdot \varphi\ lr + 2\\
&&& \text{because } \varphi\ A = \varphi\ A'\\
&< \varphi\ A + \varphi\ A_2BC + \varphi\ BC - 3 \cdot \varphi\ lr + 2 && \text{because } \varphi\ A < \varphi\ AB\\
&< \varphi\ A_2BC + 2 \cdot \varphi\ ABC - 3 \cdot \varphi\ lr + 1\\
&&& \text{because } 1 + \mathit{lg}\ x + \mathit{lg}\ y < 2 \cdot \mathit{lg}\ (x + y) \text{ if } x,y > 0\\
&< 3 \cdot (\varphi\ ABC - \varphi\ lr) + 1 && \text{because } \varphi\ A_2BC < \varphi\ ABC
\end{aligned}
$$

Now we consider the zig-zag case:



This is the case where $a < x < b$ and $B \neq \langle\rangle$. On the left we have the input and on the right the output of $\mathit{splay}\ x$. Because $B \neq \langle\rangle$, $\mathit{splay}\ x\ B = \langle B_1,\ x',\ B_2\rangle =: B'$ for some B_1, x' and B_2. The intermediate tree is obtained by replacing B by B'. The proof is very similar to the zig-zig case, the same naming conventions apply and we omit some details:

$$
\begin{aligned}
& A_{\mathit{splay}}\ x\ ABC = T_{\mathit{splay}}\ x\ A + 1 + \Phi\ AB_1B_2C - \Phi\ ABC \\
&= A_{\mathit{splay}}\ x\ B + \varphi\ AB_1 + \varphi\ B_2C - \varphi\ AB - \varphi\ B' + 1 \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{using } \varphi\ AB_1B_2C = \varphi\ ABC \\
&\leq 3 \cdot \varphi\ B + \varphi\ AB_1 + \varphi\ B_2C - \varphi\ AB - \varphi\ B' - 3 \cdot \varphi\ \mathit{lr} + 2 \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{by IH and } \mathit{lr} \in \mathit{subtrees}\ B \\
&= 2 \cdot \varphi\ B + \varphi\ AB_1 + \varphi\ B_2C - \varphi\ AB - 3 \cdot \varphi\ \mathit{lr} + 2 \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{because } \varphi\ B = \varphi\ B' \\
&< \varphi\ B + \varphi\ AB_1 + \varphi\ B_2C - 3 \cdot \varphi\ \mathit{lr} + 2 \qquad \text{because } \varphi\ B < \varphi\ AB \\
&< \varphi\ B + 2 \cdot \varphi\ ABC - 3 \cdot \varphi\ \mathit{lr} + 1 \\
&\qquad\qquad\qquad \text{because } 1 + \mathit{lg}\ x + \mathit{lg}\ y < 2 \cdot \mathit{lg}\ (x + y) \text{ if } x,y > 0 \\
&< 3 \cdot (\varphi\ ABC - \varphi\ \mathit{lr}) + 1 \qquad \text{because } \varphi\ B < \varphi\ ABC \qquad \square
\end{aligned}
$$

Because $\varphi\ \langle l,\ x,\ r\rangle \geq 1$, the above theorem implies

Corollary 21.2. $\mathit{bst}\ t \wedge x \in \mathit{set\_tree}\ t \longrightarrow A_{\mathit{splay}}\ x\ t \leq 3 \cdot (\varphi\ t - 1) + 1$

If x is not in the tree we show that there is a y in the tree such that splaying with y would produce the same tree in the same time:

Lemma 21.3. $t \neq \langle\rangle \wedge \mathit{bst}\ t \longrightarrow$
$(\exists y \in \mathit{set\_tree}\ t.\ \mathit{splay}\ y\ t = \mathit{splay}\ x\ t \wedge T_{\mathit{splay}}\ y\ t = T_{\mathit{splay}}\ x\ t)$

Element y is the last element in the tree that the search for x encounters before it hits a leaf. Naturally, the proof is by induction on the computation of splay.

Combining this lemma with Corollary 21.2 yields the final unconditional amortized complexity of splay on BSTs:

Corollary 21.4. $\mathit{bst}\ t \longrightarrow A_{\mathit{splay}}\ x\ t \leq 3 \cdot \varphi\ t + 1$

The "$-$ 1" has disappeared to accommodate the case $t = \langle\rangle$.

The amortized analysis of insertion is straightforward now. From the amortized complexity of *splay* it follows that

Lemma 21.5. $\mathit{bst}\ t \longrightarrow T_{\mathit{insert}}\ x\ t + \Phi\ (\mathit{insert}\ x\ t) - \Phi\ t \leq 4 \cdot \varphi\ t + 2$

We omit the proof which is largely an exercise in simple algebraic manipulations.

The amortized analysis of deletion is similar but a bit more complicated because of the additional function *splay_max* whose amortized running time is defined as usual:

$$A_{\mathit{splay\_max}}\ t = T_{\mathit{splay\_max}}\ t + \Phi\ (\mathit{splay\_max}\ t) - \Phi\ t$$

Like in the analysis of A_{splay}, an inductive proof yields

$$t \neq \langle\rangle \longrightarrow A_{\mathit{splay\_max}}\ t \leq 3 \cdot (\varphi\ t - 1) + 1$$

from which

$$A_{\mathit{splay\_max}}\ t \leq 3 \cdot \varphi\ t + 1$$

follows by a simple case analysis. The latter proposition, together with Corollary 21.4, proves the amortized logarithmic complexity of *delete*

$$\mathit{bst}\ t \longrightarrow T_{\mathit{delete}}\ a\ t + \Phi\ (\mathit{delete}\ a\ t) - \Phi\ t \leq 6 \cdot \varphi\ t + 2$$

in much the same way as for *insert* (Lemma 21.5).

A running time analysis of *isin* is trivial because *isin* is just *splay* followed by a constant-time test.

21.4 Exercises

Exercise 21.1. Find a sequence of numbers $n_1, n_2, \ldots n_k$ such that the insertion of theses numbers one by one creates a splay tree of height k.

Chapter Notes

Splay trees were invented and analyzed by Sleator and Tarjan [1985] for which they received the 1999 ACM Paris Kanellakis Theory and Practice Award [Kanellakis]. In addition to the amortized complexity as shown above they proved that splay trees perform as well as static BSTs (the Static Optimality Theorem) and conjectured that, roughly speaking, they even perform as well as any other BST-based algorithm. This Dynamic Optimality Conjecture is still open.

This chapter is based on earlier publications [Nipkow 2015, 2016, Nipkow and Brinkop 2019, Schoenmakers 1993].

22 Skew Heaps

Tobias Nipkow

Skew heaps are heaps in the sense of Section 14.1 and implement mergeable priority queues. Skew heaps can be viewed as a self-adjusting form of leftist heaps that attempt to maintain balance by unconditionally swapping all nodes on the merge path when merging two heaps.

22.1 Implementation of ADT *Priority_Queue_Merge*

The central operation is *merge*:

$$\mathit{merge} :: \mathit{'a\ tree} \Rightarrow \mathit{'a\ tree} \Rightarrow \mathit{'a\ tree}$$

$$\mathit{merge}\ \langle\rangle\ t = t$$

$$\mathit{merge}\ t\ \langle\rangle = t$$

$$\mathit{merge}\ (\langle l_1,\ a_1,\ r_1\rangle =: t_1)\ (\langle l_2,\ a_2,\ r_2\rangle =: t_2)$$
$$= (\textbf{if}\ a_1 \le a_2\ \textbf{then}\ \langle \mathit{merge}\ t_2\ r_1,\ a_1,\ l_1\rangle\ \textbf{else}\ \langle \mathit{merge}\ t_1\ r_2,\ a_2,\ l_2\rangle)$$

The remaining operations (*empty*, *insert*, *get_min* and *del_min*) are defined as in Section 14.1.

The following properties of *merge* have easy inductive proofs:

$$|\mathit{merge}\ t_1\ t_2| = |t_1| + |t_2|$$

$$\mathit{mset\_tree}\ (\mathit{merge}\ t_1\ t_2) = \mathit{mset\_tree}\ t_1 + \mathit{mset\_tree}\ t_2$$

$$\mathit{heap}\ t_1 \wedge \mathit{heap}\ t_2 \longrightarrow \mathit{heap}\ (\mathit{merge}\ t_1\ t_2)$$

Now it is straightforward to prove the correctness of the implementation w.r.t. the ADT *Priority_Queue_Merge*.

Skew heaps attempt to maintain balance, but this does not always work:

Exercise 22.1. Find a sequence of numbers $n_1,\ n_2,\ \ldots n_k$ such that the insertion of these numbers one by one creates a tree of height k. Prove that this sequence will produce a tree of height k.

Nevertheless, insertion and deletion have amortized logarithmic complexity.

22.2 Amortized Analysis

The key is the definition of the potential. It counts the number of **right-heavy** (*rh*) nodes:

$\Phi :: {}'a\ tree \Rightarrow int$

$\Phi\ \langle\rangle = 0$
$\Phi\ \langle l, \_, r\rangle = \Phi\ l + \Phi\ r + \mathit{rh}\ l\ r$

$\mathit{rh} :: {}'a\ tree \Rightarrow {}'a\ tree \Rightarrow nat$

$\mathit{rh}\ l\ r = (\mathbf{if}\ |l| < |r|\ \mathbf{then}\ 1\ \mathbf{else}\ 0)$

The rough intuition: because *merge* descends along the right spine, the more right-heavy nodes a tree contains, the longer *merge* takes.

Two auxiliary functions count the number of right-heavy nodes on the left spine (*lrh*) and left-heavy (= not right-heavy) nodes on the right spine (*rlh*):

$\mathit{lrh} :: {}'a\ tree \Rightarrow nat$

$\mathit{lrh}\ \langle\rangle = 0$
$\mathit{lrh}\ \langle l, \_, r\rangle = \mathit{rh}\ l\ r + \mathit{lrh}\ l$

$\mathit{rlh} :: {}'a\ tree \Rightarrow nat$

$\mathit{rlh}\ \langle\rangle = 0$
$\mathit{rlh}\ \langle l, \_, r\rangle = 1 - \mathit{rh}\ l\ r + \mathit{rlh}\ r$

The following properties have automatic inductive proofs:

$$2^{\mathit{lrh}\ t} \le |t| + 1 \qquad 2^{\mathit{rlh}\ t} \le |t| + 1$$

They imply

$$\mathit{lrh}\ t \le \mathit{lg}\ |t|_1 \qquad \mathit{rlh}\ t \le \mathit{lg}\ |t|_1 \tag{22.1}$$

Now we are ready for the amortized analysis. All time functions can be found in Appendix B.10. The key lemma is an upper bound of the amortized complexity of *merge* in terms of *lrh* and *rlh*:

Lemma 22.1. $T_{\mathit{merge}}\ t_1\ t_2 + \Phi\ (\mathit{merge}\ t_1\ t_2) - \Phi\ t_1 - \Phi\ t_2$
$\le \mathit{lrh}\ (\mathit{merge}\ t_1\ t_2) + \mathit{rlh}\ t_1 + \mathit{rlh}\ t_2 + 1$

Proof by induction on the computation of *merge*. We consider only the node-node case: let $t_1 = \langle l_1, a_1, r_1\rangle$ and $t_2 = \langle l_2, a_2, r_2\rangle$. W.l.o.g. assume $a_1 \le a_2$. Let $m =$ *merge* $t_2\ r_1$.

$$\begin{aligned}
&T_{\mathit{merge}}\ t_1\ t_2 + \Phi\ (\mathit{merge}\ t_1\ t_2) - \Phi\ t_1 - \Phi\ t_2\\
&= T_{\mathit{merge}}\ t_2\ r_1 + 1 + \Phi\ m + \Phi\ l_1 + \mathit{rh}\ m\ l_1 - \Phi\ t_1 - \Phi\ t_2\\
&= T_{\mathit{merge}}\ t_2\ r_1 + 1 + \Phi\ m + \mathit{rh}\ m\ l_1 - \Phi\ r_1 - \mathit{rh}\ l_1\ r_1 - \Phi\ t_2\\
&\le \mathit{lrh}\ m + \mathit{rlh}\ t_2 + \mathit{rlh}\ r_1 + \mathit{rh}\ m\ l_1 + 2 - \mathit{rh}\ l_1\ r_1 && \text{by IH}\\
&= \mathit{lrh}\ m + \mathit{rlh}\ t_2 + \mathit{rlh}\ t_1 + \mathit{rh}\ m\ l_1 + 1\\
&= \mathit{lrh}\ (\mathit{merge}\ t_1\ t_2) + \mathit{rlh}\ t_1 + \mathit{rlh}\ t_2 + 1 && \square
\end{aligned}$$

As a consequence we can prove the following logarithmic upper bound on the amortized complexity of *merge*:

$$\begin{aligned}
&T_{\mathit{merge}}\ t_1\ t_2 + \Phi\ (\mathit{merge}\ t_1\ t_2) - \Phi\ t_1 - \Phi\ t_2\\
&\le \mathit{lrh}\ (\mathit{merge}\ t_1\ t_2) + \mathit{rlh}\ t_1 + \mathit{rlh}\ t_2 + 1 && \text{by Lemma 22.1}\\
&\le \mathit{lg}\ |\mathit{merge}\ t_1\ t_2|_1 + \mathit{lg}\ |t_1|_1 + \mathit{lg}\ |t_2|_1 + 1 && \text{by (22.1)}\\
&\le \mathit{lg}\ (|t_1|_1 + |t_2|_1 - 1) + \mathit{lg}\ |t_1|_1 + \mathit{lg}\ |t_2|_1 + 1\\
& && \text{because } |\mathit{merge}\ t_1\ t_2| = |t_1| + |t_2|\\
&\le \mathit{lg}\ (|t_1|_1 + |t_2|_1) + 2 \cdot \mathit{lg}\ (|t_1|_1 + |t_2|_1) + 1\\
&= 3 \cdot \mathit{lg}\ (|t_1|_1 + |t_2|_1) + 1
\end{aligned}$$

The amortized complexities of insertion and deletion follow easily from the complexity of *merge*:

$$\begin{aligned}
&T_{\mathit{insert}}\ a\ t + \Phi\ (\mathit{insert}\ a\ t) - \Phi\ t \le 3 \cdot \mathit{lg}\ (|t|_1 + 2) + 1\\
&T_{\mathit{del\_min}}\ t + \Phi\ (\mathit{del\_min}\ t) - \Phi\ t \le 3 \cdot \mathit{lg}\ (|t|_1 + 2) + 1
\end{aligned}$$

Chapter Notes

Skew heaps were invented by Sleator and Tarjan [1986] as one of the first self-organizing data structures. Their presentation was imperative. Our presentation follows earlier work by Nipkow [2015] and Nipkow and Brinkop [2019] based on the functional account by Kaldewaij and Schoenmakers [1991].

23 Pairing Heaps

Tobias Nipkow

The pairing heap is another form of a self-adjusting priority queue.

23.1 Implementation

A **pairing heap** is a heap in the sense that it is a tree with the minimal element at the root — except that it is not a binary tree but a tree where each node has a list of children:

```
datatype 'a hp = Hp 'a ('a hp list)

type_synonym 'a heap = 'a hp option
```

To accommodate the empty heap, we have put *option* on top. We could have avoided the *option* layer by defining **datatype** *'a hp* = *Empty* | *Hp 'a* (*'a hp list*). The drawback of this one-step definition is that *Empty* may occur inside a non-*Empty* *hp*. The amortized analysis needs to rule out such ill-formed heaps, i.e. it requires an invariant, something we can avoid altogether (at the expense of two types rather than one). The invariants and abstraction functions follow the heap paradigm:

```
php :: 'a hp ⇒ bool
php (Hp x hs) = (∀h∈set hs. (∀y∈#mset_hp h. x ≤ y) ∧ php h)

invar :: 'a heap ⇒ bool
invar ho = (case ho of None ⇒ True | Some h ⇒ php h)

mset_hp :: 'a hp ⇒ 'a multiset
mset_hp (Hp x hs) = ⦃x⦄ + sum_list (map mset_hp hs)

mset_heap :: 'a heap ⇒ 'a multiset
mset_heap ho = (case ho of None ⇒ ⦃⦄ | Some h ⇒ mset_hp h)
```

The implementations of *empty* and *get_min* are obvious, and *insert* follows the standard heap paradigm:

```
empty = None

get_min :: 'a heap ⇒ 'a
get_min (Some (Hp x _)) = x

insert :: 'a ⇒ 'a heap ⇒ 'a heap
insert x None = Some (Hp x [])
insert x (Some h) = Some (link (Hp x []) h)

link :: 'a hp ⇒ 'a hp ⇒ 'a hp
link (Hp x1 hs1) (Hp x2 hs2)
= (if x1 < x2 then Hp x1 (Hp x2 hs2 # hs1) else Hp x2 (Hp x1 hs1 # hs2))
```

Auxiliary function *link* simply adds one of the two heaps to the front of the other, depending on the root values.

Function *merge* is not recursive but delegates to *link*:

```
merge :: 'a heap ⇒ 'a heap ⇒ 'a heap
merge ho None = ho
merge None ho = ho
merge (Some h1) (Some h2) = Some (link h1 h2)
```

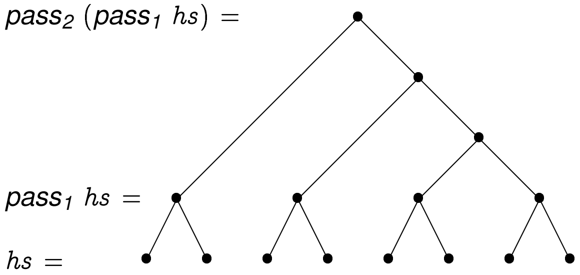
Thus *merge* and *insert* have constant running time. All the work is offloaded on *del_min* which delegates to a 2-pass algorithm:

```
del_min :: 'a heap ⇒ 'a heap
del_min None = None
del_min (Some (Hp _ hs)) = pass2 (pass1 hs)

pass1 :: 'a hp list ⇒ 'a hp list
pass1 (h1 # h2 # hs) = link h1 h2 # pass1 hs
pass1 hs = hs
```

$$\mathit{pass}_2 :: \textit{'a hp list} \Rightarrow \textit{'a heap}$$

$$\mathit{pass}_2\ [] = \mathit{None}$$
$$\mathit{pass}_2\ (h \mathbin{\#} hs) = \mathit{Some}\ (\mathbf{case}\ \mathit{pass}_2\ hs\ \mathbf{of}\ \mathit{None} \Rightarrow h \mid \mathit{Some}\ h' \Rightarrow \mathit{link}\ h\ h')$$

The following diagram exemplifies both passes:



Pass 1 links pairs of adjacent hps (hence the name **pairing heap**) and pass 2 links the resulting list of hps in a cascade into a single heap.

Clearly *del_min* can take linear time but it will turn out that the constant-time *insert* saves enough to guarantee amortized logarithmic complexity for both insertion and deletion.

Comparing pairing heaps and binomial heaps and forests we find: Type hp is almost identical to type *tree* in the representation of binomial heaps and function *link* is almost identical to its namesake on binomial heaps. However, *insert* and *merge* are constant-time, in contrast to their namesakes on binomial forests.

Exercise 23.1. The composition of $pass_1$ and $pass_2$ has the drawback of creating an intermediate list. Define a single-pass function *merge_pairs* that behaves like $pass_2 \circ pass_1$ and is no slower but does not create an intermediate list. Prove

$$\mathit{merge\_pairs}\ hs = \mathit{pass}_2\ (\mathit{pass}_1\ hs)$$
$$T_{\mathit{merge\_pairs}}\ hs \le T_{\mathit{pass}_1}\ hs + T_{\mathit{pass}_2}\ (\mathit{pass}_1\ hs)$$

23.1.1 Correctness

The properties in the specifications *Priority_ Queue(_ Merge)* are easily established. Function *del_min* requires the following lemmas (all proofs are routine inductions):

$$ho \neq \mathit{None} \longrightarrow \mathit{mset\_heap}\ (\mathit{del\_min}\ ho) = \mathit{mset\_heap}\ ho - \{\!\{\mathit{get\_min}\ ho\}\!\}$$
$$ho \neq \mathit{None} \longrightarrow \mathit{get\_min}\ ho \in_{\#} \mathit{mset\_hp}\ (\mathit{the}\ ho)$$
$$ho \neq \mathit{None} \wedge \mathit{invar}\ ho \wedge x \in_{\#} \mathit{mset\_hp}\ (\mathit{the}\ ho) \longrightarrow \mathit{get\_min}\ ho \le x$$
$$\mathit{invar}\ ho \longrightarrow \mathit{invar}\ (\mathit{del\_min}\ ho)$$

23.2 Amortized Analysis

The potential function Φ is defined in terms of a size function. More precisely, we need size functions for the three types under consideration: *'a hp list*, *'a hp* and *'a heap*. For readability we defined three instances of an overloaded function *sz*:

```
sz :: 'a hp list ⇒ nat
sz (Hp x hsl # hsr) = sz hsl + sz hsr + 1
sz [] = 0

sz :: 'a hp ⇒ nat
sz h = sz (hps h) + 1

sz :: 'a heap ⇒ nat
sz ≡ lift_hp 0 sz

lift_hp :: 'b ⇒ ('a hp ⇒ 'b) ⇒ 'a heap ⇒ 'b
lift_hp c _ None = c
lift_hp _ f (Some h) = f h

hps :: 'a hp ⇒ 'a hp list
hps (Hp _ hs) = hs
```

Function *sz* essentially just counts the number of constructors.

The potential function Φ is overloaded in the same way:

```
Φ :: 'a hp list ⇒ real
Φ [] = 0
Φ (Hp x hsl # hsr) = Φ hsl + Φ hsr + lg (sz hsl + sz hsr + 1)

Φ :: 'a hp ⇒ real
Φ h = Φ (hps h) + lg (sz (hps h) + 1)

Φ :: 'a heap ⇒ real
Φ ≡ lift_hp 0 Φ
```

These definitions may look a bit mysterious. Section 23.3 shows how they follow from a simple uniform definition where heaps are represented by binary trees.

It is straightforward to prove that the non-recursive *insert* and *merge* have amortized logarithmic complexity:

$$T_{insert}\ a\ ho + \Phi\ (insert\ a\ ho) - \Phi\ ho \leq lg\ (sz\ ho + 1)$$

$$T_{merge}\ ho_1\ ho_2 + \Phi\ (merge\ ho_1\ ho_2) - \Phi\ ho_1 - \Phi\ ho_2$$
$$\leq 2 \cdot lg\ (sz\ ho_1 + sz\ ho_2 + 1)$$

The analysis of *del_min* is more work. Its running time on *Some* h is linear in the length of *hps* h. Therefore we have to show that the potential change compensates for this linear work. Our main goal is this:

Theorem 23.1. $\Phi\ (del\_min\ (Some\ h)) - \Phi\ (Some\ h)$
$\leq 2 \cdot lg\ (sz\ (hps\ h) + 1) - |hps\ h| + 2$

We will prove it in two steps: First we show that *pass₁* frees enough potential to compensate for the work linear in $|hs|$ and increases the potential only by a logarithmic term. Then we show that the increase due to *pass₂* is also only at most logarithmic. Combining these results one easily shows that the amortized running time of *del_min* is indeed logarithmic.

First we analyze the potential difference caused by $pass_1$:

Lemma 23.2. $\Phi\ (pass_1\ hs) - \Phi\ hs \leq 2 \cdot lg\ (sz\ hs + 1) - |hs| + 2$

Proof by induction on the computation of $pass_1$. The base cases are trivial. We focus on the induction step. Let $hs' = h_1\ \#\ h_2\ \#\ hs$, $h_1 = Hp\ \_\ hs_1$, $h_2 = Hp\ \_\ hs_2$, $n_1 = sz\ hs_1$, $n_2 = sz\ hs_2$ and $m = sz\ hs$.

$$\begin{aligned}
&\Phi\ (pass_1\ hs') - \Phi\ hs' \\
&= lg\ (n_1 + n_2 + 1) - lg\ (n_2 + m + 1) + \Phi\ (pass_1\ hs) - \Phi\ hs \\
&\leq lg\ (n_1 + n_2 + 1) - lg\ (n_2 + m + 1) + 2 \cdot lg\ (m + 1) - |hs| + 2 \quad \text{by IH} \\
&\leq 2 \cdot lg\ (n_1 + n_2 + m + 1) - lg\ (n_2 + m + 1) + lg\ (m + 1) - |hs| \\
&\qquad \text{because } lg\ x + lg\ y + 2 \leq 2 \cdot lg\ (x + y) \text{ if } x,y > 0 \\
&\leq 2 \cdot lg\ (n_1 + n_2 + m + 2) - |hs| \\
&= 2 \cdot lg\ (sz\ hs') - |hs'| + 2 \\
&\leq 2 \cdot lg\ (sz\ hs' + 1) - |hs'| + 2
\end{aligned}$$
□

Now we turn to $pass_2$:

Lemma 23.3. $hs \neq [] \longrightarrow \Phi\ (pass_2\ hs) - \Phi\ hs \leq lg\ (sz\ hs)$

Proof by induction on hs. The base case is trivial. The induction step $Hp\ \_\ hs_1\ \#\ hs$ is trivial if $hs = []$. We assume $hs \neq []$. Thus $pass_2\ hs = Some\ (Hp\ \_\ hs_2)$ for some hs_2. We also need that for all hs

$$\mathit{sz}\ (\mathit{pass}_2\ hs) = \mathit{sz}\ hs$$

The proof is a straightforward induction on hs. This implies $\mathit{sz}\ hs = \mathit{sz}\ hs_2 + 1$. Moreover, by definition of *link* we have

$$\Phi\ (\mathit{link}\ h_1\ h_2) = \Phi\ hs_1 + \Phi\ hs_2 + \mathit{lg}\ (n_1 + n_2 + 1) + \mathit{lg}\ (n_1 + n_2 + 2) \quad (*)$$

Finally note that the IH $hs \neq [] \longrightarrow \Phi\ (\mathit{pass}_2\ hs) - \Phi\ hs \leq \mathit{lg}\ (\mathit{sz}\ hs)$ reduces to

$$\Phi\ hs_2 - \Phi\ hs \leq 0 \qquad (**)$$

The overall claim follows:

$$\begin{aligned}
&\Phi\ (\mathit{pass}_2\ (h_1\ \#\ hs)) - \Phi\ (h_1\ \#\ hs) \\
&= \Phi\ (\mathit{link}\ h_1\ h_2) - (\Phi\ hs_1 + \Phi\ hs + \mathit{lg}\ (n_1 + \mathit{sz}\ hs + 1)) \\
&= \Phi\ hs_2 + \mathit{lg}\ (n_1 + n_2 + 1) - \Phi\ hs && \text{by } (*) \\
&\leq \mathit{lg}\ (n_1 + n_2 + 1) && \text{by } (**) \\
&\leq \mathit{lg}\ (\mathit{sz}\ (h_1\ \#\ hs)) && \square
\end{aligned}$$

Corollary 23.4. $\Phi\ (\mathit{pass}_2\ hs) - \Phi\ hs \leq \mathit{lg}\ (\mathit{sz}\ hs + 1)$

Theorem 23.1 follows easily:

$$\begin{aligned}
&\Phi\ (\mathit{del\_min}\ (\mathit{Some}\ h)) - \Phi\ (\mathit{Some}\ h) \\
&= \Phi\ (\mathit{pass}_2\ (\mathit{pass}_1\ hs)) - (\mathit{lg}\ (\mathit{sz}\ hs + 1) + \Phi\ hs) && \text{where } h = \mathit{Hp}\ \_\ hs \\
&\leq \Phi\ (\mathit{pass}_1\ hs) - \Phi\ hs && \text{by Corollary 23.4} \\
&\leq 2 \cdot \mathit{lg}\ (\mathit{sz}\ hs + 1) - |hs| + 2 && \text{by Lemma 23.2}
\end{aligned}$$

Combining the following inductive upper bound for the running time of the two passes

$$T_{\mathit{pass}_2}\ (\mathit{pass}_1\ hs) + T_{\mathit{pass}_1}\ hs \leq 2 + |hs|$$

with Theorem 23.1 yields the third and final amortized running time:

$$T_{\mathit{del\_min}}\ ho + \Phi\ (\mathit{del\_min}\ ho) - \Phi\ ho \leq 2 \cdot \mathit{lg}\ (\mathit{sz}\ ho + 1) + 4$$

Thus we have proved that insertion, merging and deletion all have amortized logarithmic running times.

23.3 Pairing Heaps as Trees

Pairing heaps can be represented as binary trees as follows: a heap $\mathit{Hp}\ x\ hs$ is represented by the tree $\langle \mathit{trees}\ hs,\ x,\ \langle\rangle\rangle$ where

```
trees :: 'a hp list ⇒ 'a tree

trees [] = ⟨⟩
trees (Hp x lhs # rhs) = ⟨trees lhs, x, trees rhs⟩
```

None is represented by $\langle\rangle$ and *Some* is dropped. Although it is like working with untyped LISP S-expressions, it has the big advantage that we now have to deal only with a single type, trees. This is particularly relevant for the amortized analysis, where a single size and potential function suffice. In fact, the size function is simply the standard size function on trees and Φ is (almost) the potential function used for splay trees:

$$\Phi :: {}'a\ tree \Rightarrow real$$

$$\Phi\ \langle\rangle = 0$$

$$\Phi\ \langle l,\ x,\ r\rangle = \Phi\ l + \Phi\ r + \mathit{lg}\ |\langle l,\ x,\ r\rangle|$$

The tree representation simplifies both the analysis and the implementation. Conversely, via the above mapping *trees* from heaps to trees we can derive the definitions of *sz* and Φ in Section 23.2 from the definitions of *size* and Φ on trees. For example, from the (alternative) definition $\mathit{sz}\ hs = |\mathit{trees}\ hs|$, the two defining equations for *sz* on $'a\ hp\ list$ in Section 23.2 follow directly from the definition of *trees*.

Chapter Notes

Pairing heaps were invented by Fredman et al. [1986] as a simpler but competitive alternative to Fibonacci heaps. The authors gave the amortized analysis presented above (but using binary trees as sketched in Section 23.3) and conjectured that it can be improved. Later research confirmed this [Iacono 2000, Iacono and Yagnatinsky 2016, Pettie 2005] but the final analysis is still open. An empirical study [Larkin et al. 2014] showed that pairing heaps do indeed outperform Fibonacci heaps in practice. This chapter is based on an article by Nipkow and Brinkop [2019].

Part V

Selected Topics

24 Graph Algorithms

Mohammad Abdulaziz

Graphs are a fundamental structure in mathematics and computer science, and algorithms processing them span some of the most basic in computer science, like the ones we will discuss here, up to some of the deepest, like algorithms for matching and other combinatorial optimisation problems. Indeed, much of this very book is dedicated to studying trees, which are a certain type of graphs, and their application in storing and manipulating data. In this chapter we focus on a more general class of graphs, namely, directed graphs, algorithms for processing them, and formal reasoning about those algorithms, in which we prove, using a theorem prover, desired properties of those algorithms.

The first step in the process of reasoning about graph algorithms is that of representing or modelling a (directed) graph in a theorem prover. Earlier in the book, for instance, graphs were represented using weight mappings, where an infinite weight indicates a lack of an edge. Here we choose a different model of directed graphs: a **directed graph** is a set of pairs, each of which is modelling an edge, formally, $('v \times 'v)$ *set*. This model emphasises the view of a graph as a set, which makes automatic reasoning easier as it glosses over implementation details. For such graphs, we define a number of auxiliary functions and predicates to enable reasoning about them. These are

- A function returning the set of vertices in a directed graph:

$$\mathit{dVs} :: ('v \times 'v)\ \mathit{set} \Rightarrow 'v\ \mathit{set}$$
$$\mathit{dVs}\ G = \bigcup\ \{\{v_1,\ v_2\} \mid (v_1,\ v_2) \in G\}$$

- A function returning the **neighbourhood** of a vertex

$$\mathit{neighbourhood} :: ('v \times 'v)\ \mathit{set} \Rightarrow 'v \Rightarrow 'v\ \mathit{set}$$
$$\mathit{neighbourhood}\ G\ u = \{v \mid (u,\ v) \in G\}$$

- A predicate indicating that a list of vertices forms a **walk** in the graph:

```
vwalk :: ('v × 'v) set ⇒ 'v list ⇒ bool
vwalk _ []
vwalk E [v] = (v ∈ dVs E)
vwalk E (u # v # vs) = ((u, v) ∈ E ∧ vwalk E (v # vs))
```

- An auxiliary predicate indicating that a list of vertices constitutes a walk between two given vertices:

```
vwalk_bet :: ('v × 'v) set ⇒ 'v ⇒ 'v list ⇒ 'v ⇒ bool
vwalk_bet G u p v = (vwalk G p ∧ p ≠ [] ∧ hd p = u ∧ last p = v)
```

Although there is a myriad of other properties that could be defined for directed graphs, the ones we defined above are enough for our purposes for now as we will mainly be studying algorithms that reason about reachability between vertices in a given directed graph.

We note that, although this book is mainly about executable algorithms, this representation of graphs cannot be used for specifying executable algorithms. It glosses over implementation details, making it more suitable for mathematical reasoning. On the other hand it is not guaranteed to be finite, which complicates any computational interpretation of the type.

24.1 Depth-First Search

The first algorithm we will consider here is **depth-first** search (DFS). DFS is a so-called **graph-traversal** algorithm, which is a class of algorithms that process vertices of a directed graph in a given traversal order. For DFS, that order is, as one could guess from the name, depth-first. This means that, while processing a vertex, the algorithm processes one neighbouring vertex and all its descendants, before moving on to any of its other neighbours. Figure 24.1 shows a number of depth-first traversals. In its simplest form, such a traversal has the goal of finding a vertex-walk between a given source vertex (called s henceforth) and a target vertex (called t henceforth). In particular, we would like an implementation of DFS to satisfy one property: it finds a vertex-walk iff there is one.

24.1.1 Modelling Graphs: an Algorithmic Perspective

Now, as we have a general understanding of what is required from DFS, we start with the specifics of implementing and reasoning about DFS. The first aspect is

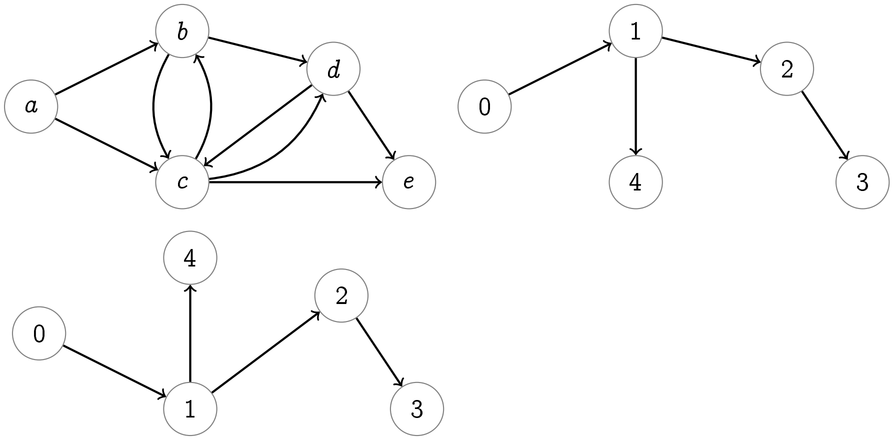


Figure 24.1 A directed graph, and illustrations of two of its depth-first traversals rooted at vertex a. In each of those, vertices are numbered according to the 'time' at which they had been traversed. Only traversed edges are shown in the latter two graphs.

modelling directed graphs. Recall that we have already provided a formal model of directed graphs – using which one can formally prove any result on directed graphs. Nonetheless, that model of directed graphs does not immediately allow executability. For instance, a common operation in graph algorithms is picking a neighbour of a vertex and then processing it. In that representation it is not immediately obvious how this could be implemented because the set of neighbours is a mathematical set with no notion of ordering that allows one to deterministically pick a neighbour. One naive way to handle such nondeterminism is to use lists as a representation of sets, i.e. the neighbourhood of a vertex would be a list of vertices, and the graph itself would be a list of pairs, and so on. This approach would solve the problem of nondeterminism, as choosing a neighbour, for instance, could be achieved by taking the head of the list of neighbours. However, this approach presents two problems:

1. It fixes an implementation of sets, which is inflexible and slower than it needs to be. E.g. finding whether a vertex is in the neighbourhood of another vertex can be done in time linear, in the worst case, in the size of the neighbourhood. This is much worse than the logarithmic time achievable if the set is represented efficiently by a tree.
2. Proving graph-theoretic facts about graphs represented as lists can be cumbersome and adds an unneeded layer of complexity.

ADT $Set\_\ Choose = Set\ +$

interface

$sel :: {}'s \Rightarrow {}'a$

specification

$s \neq empty \longrightarrow isin\ s\ (sel\ s)$ $\qquad (choose)$

Figure 24.2 ADT *Set_ Choose*

To solve the first problem, we follow the approach of Abstract Data Types (ADTs) employed earlier in this book to parametrically specify ***DFS***, where we parameterise it over an efficient representation of a map, used for adjacency, and a set, used to represent neighbourhoods of vertices. For that, we need two ADTs. The first ADT is that of sets with a choice operator, i.e. a function that returns an arbitrary element of the set, if the set is not empty. This ADT has the same interface as that of *Set* from Chapter 6, but with one additional function that selects an arbitrary element of the set, if the set is not empty. The ADT is shown in Figure 24.2.

The next ADT here represents the graph that is to be processed by an algorithm. The details of that ADT are shown in Figure 24.3. We note a number of points. First, this ADT does not introduce any new operations of its own; it is merely an ADT that uses and renames the operations of the *Map* ADT from Chapter 6 and the *Set_ Choose* ADT, where the latter is used to model neighbourhoods of vertices and the former is an adjacency map mapping every vertex to its neighbourhood. Because we introduce no new operations, we do not need new specifications or abstraction functions. However, we need to take care of two things: 1. to make sure that the types of the two ADTs are consistent, e.g. the adjacency map maps vertices to neighbourhoods of the same type as the type *Set_ Choose*; and 2. to make sure that constants in the interfaces of the two used ADTs do not have the same names.

We note that the two ADTs do not provide direct access to crucial operations needed for manipulating graphs, like adding edges to a graph. Such operations can be implemented as shown below, in terms of the ADTs' interfaces:

```
add_edge :: $'adjmap \Rightarrow {}'v \Rightarrow {}'v \Rightarrow {}'adjmap$
add_edge $G\ u\ v$
$=$ (case $lookup\ G\ u$ of
    None $\Rightarrow$
      let $vset' = insert\ v\ \emptyset_V;\ digraph' = update\ u\ vset'\ G$ in $digraph'$
```

ADT $Pair\_Graph\_Specs = adjmap\colon Map + vset\colon Set\_Choose +$

interface

$\emptyset_G :: {}'adjmap$ (**renaming** $adjmap.empty$)
$update :: {}'v \Rightarrow {}'vset \Rightarrow {}'adjmap \Rightarrow {}'adjmap$ (**renaming** $adjmap.update$)
$lookup :: {}'adjmap \Rightarrow {}'v \Rightarrow {}'vset\ option$ (**renaming** $adjmap.lookup$)
$adjmap\_inv :: {}'adjmap \Rightarrow bool$ (**renaming** $adjmap.inv$)

$\emptyset_V :: {}'vset$ (**renaming** $vset.empty$)
$insert :: {}'v \Rightarrow {}'vset \Rightarrow {}'vset$ (**renaming** $vset.insert$)
$isin :: {}'vset \Rightarrow {}'v \Rightarrow bool$ (**renaming** $vset.isin$)
$t\_set :: {}'vset \Rightarrow {}'v\ set$ (**renaming** $vset.set$)
$vset\_inv :: {}'vset \Rightarrow bool$ (**renaming** $vset.inv$)
$sel :: {}'vset \Rightarrow {}'v$ (**renaming** $vset.sel$)

Figure 24.3 ADT *Pair_Graph_Specs*. Note: **renaming** indicates that, although the new ADT extends existing ADTs, it will use a different name to refer to members of the interface of the ADT it extends.

```
    | Some $vset \Rightarrow$
        let $vset = \mathit{the}\ (lookup\ G\ u);\ vset' = insert\ v\ vset;$
            $digraph' = update\ u\ vset'\ G$
        in $digraph')$
```

```
neighb $:: {}'adjmap \Rightarrow {}'v \Rightarrow {}'vset$
$\mathcal{N}_G\ G\ v = ($case $lookup\ G\ v$ of None $\Rightarrow \emptyset_V$ | Some $vset \Rightarrow vset)$
```

The *Pair_Graph_Specs* ADT solves the first problem we mentioned above, namely, it gives us flexibility by not fixing an implementation of neighbourhoods and graphs. However, to prove facts about it, we define the following abstraction function connecting the ADT *Pair_Graph_Specs* to the more abstract representation $({}'v \times {}'v)\ set$, which is more amenable to mathematical reasoning:

```
digraph_abs $:: {}'adjmap \Rightarrow ({}'v \times {}'v)\ set$
```

$$[G]_G = \{(u,\ v) \mid v \in_G \mathcal{N}_G\ \ G\ u\}$$

Note: in the rest of this chapter, we will use [.] to denote the mathematical abstraction of a given structure. We use $[G]_G$ for graphs and $[vset]_s$ for sets.

This abstraction function is used to connect our two representations of directed graphs using the following lemmas:

$$\textsf{graph\_inv}\ G \longrightarrow (v \in [\mathcal{N}_G\ \ G\ u]_s) = ((u,\ v) \in [G]_G)$$
$$\textsf{graph\_inv}\ G \longrightarrow [\mathcal{N}_G\ \ G\ u]_s = \textsf{neighbourhood}\ [G]_G\ u$$
$$\textsf{graph\_inv}\ G \longrightarrow [\textsf{add\_edge}\ G\ u\ v]_G = \textsf{insert}\ (u,\ v)\ [G]_G$$

Note that all the above lemmas are conditional on the graph satisfying some invariant, denoted by $\textsf{graph\_inv}$. This invariant is not specified in the *Pair_ Graph_ Specs*'s specification, but rather defined in terms of invariants of *Map* and *Set_ Choose* as follows:

$$\textsf{graph\_inv} :: {}'adjmap \Rightarrow bool$$
$$\textsf{graph\_inv}\ G$$
$$= (adjmap\_inv\ G \wedge (\forall v\ vset.\ lookup\ G\ v = \textsf{Some}\ vset \longrightarrow vset\_inv\ vset))$$

Again, for the operations we defined on directed graphs, we need to know that they preserve this invariant. This is derived here from the fact that the operations in the interfaces of *Map* and *Set_ Choose* preserve the invariants of these respective ADTs.

$$\textsf{graph\_inv}\ G \longrightarrow \textsf{graph\_inv}\ (\textsf{add\_edge}\ G\ u\ v)$$
$$\textsf{graph\_inv}\ G \longrightarrow \textsf{graph\_inv}\ (\textsf{delete\_edge}\ G\ u\ v)$$

The above lemmas connecting *Pair_ Graph_ Specs* and the abstract model of graphs allow us to specify algorithms in terms of the ADT *Pair_ Graph_ Specs*, yet at the same time prove and specify properties of the algorithm in terms of the abstract model of directed graphs.

24.1.2 Modelling DFS

Now, given those two models of directed graphs and their connection, we are ready to specify and reason about graph algorithms. Although DFS can be modelled as a simple recursive functional program, we model DFS following a methodology that can scale to modelling significantly more involved iterative algorithms. The first thing we note is that the algorithm will be implemented in terms of the ADT *Pair_ Graph_ Specs*, providing a model of the graph and operations on it, and the ADT *Set2* from

ADT $DFS = Graph\colon Pair\_Graph\_Specs + set\_ops\colon Set2 +$

interface
$(\cup_G) :: {}'vset \Rightarrow {}'vset \Rightarrow {}'vset$ (**renaming** $set\_ops.union$)
$(\cap_G) :: {}'vset \Rightarrow {}'vset \Rightarrow {}'vset$ (**renaming** $set\_ops.inter$)
$(-_G) :: {}'vset \Rightarrow {}'vset \Rightarrow {}'vset$ (**renaming** $set\_ops.diff$)

$G :: {}'adjmap$
$s :: {}'v$
$t :: {}'v$

Figure 24.4 Interface of DFS. We omit the interface $Pair\_Graph\_Specs$ and $Set2$ as they are unchanged from Figure 24.3. We only layout the additional elements of the interface: the graph $G :: {}'adjmap$, the source $s :: {}'v$, the target $t :: {}'v$, and the ADT of binary set operations $(\cup_G)$, $(\cap_G)$, and $(-_G)$.

Figure 10.1, providing binary set operations. Its interface will additionally fix the graph which it processes, G, a source vertex s and a target vertex t. This is shown in Figure 24.4.

In addition to those operations, another element of modelling DFS is its program state, i.e. the local variables that would appear in an imperative presentation of the algorithm. We model the state of DFS using the following record:

record $({}'v, {}'vset)$ $DFS\_state =$ *stack* $:: {}'v\ list$
seen $:: {}'vset$
return $:: return$

The last element of the above record is an indicator as to whether the target vertex can be reached from the source vertex. It is defined as the following algebraic data type:

datatype $return$ = *Reachable* | *NotReachable*

The last remaining part is the actual implementation of DFS, which we do as follows:

```
DFS :: ('v, 'vset) DFS_state ⇒ ('v, 'vset) DFS_state
DFS dfs_state
= (case stack dfs_state of [] ⇒ dfs_state(|return := NotReachable|)
    | v # stack_tl ⇒
      if v = t then dfs_state(|return := Reachable|)
      else if 𝒩_G v -_G seen dfs_state ≠ ∅_V
           then let u = sel (𝒩_G v -_G seen dfs_state);
                    stack' = u # stack dfs_state;
                    seen' = insert u (seen dfs_state)
                in DFS (dfs_state(|stack := stack', seen := seen'|))
           else let stack' = stack_tl
                in DFS (dfs_state(|stack := stack'|)))
```

In this definition, we model a while-loop which performs DFS as a recursive function. This recursive function explicitly manipulates a program state by changing the members of the record modelling the local variables. The algorithm keeps track of the vertices it still has to process in the stack *stack* and all the vertices it finished processing in a set called *seen*. If the stack is empty, then the algorithm concludes the target cannot be reached from the source. Otherwise, vertices are processed from the top of the stack. To process a vertex, we check if it is the target. If it is, then we are done. If it is not, then we select one neighbour of the vertex and push it to the top of the stack for processing. If the current vertex has no neighbours, it is removed from the stack and added to *seen*. Note that, since G::'*adjmap* is fixed, we do not pass it as the first argument from $\mathcal{N}_G\ v$.

24.1.3 Reasoning About *DFS*

Recall that the function *DFS* is implemented as a recursive function. Thus, the most natural way to reason about it is by mathematical induction. As stated in the first chapter of this book, for programs that are not primitive recursive, reasoning is primarily done by computation induction, in which the induction principle is based on and follows the terminating computation performed by the program. Standard approaches [Krauss] can already automatically synthesise and prove such induction principles. However, for DFS, for instance, an automatically generated induction principle would have two problems. First, the induction principle will be conditional on the state we are reasoning about. In particular, we have to assume that *DFS* terminates on that state for the induction principle to be applicable. This is of course because we have not (yet) proved that the algorithm terminates in general. We thus first prove the function partially correct, by showing that the desired properties hold starting at any state from which the function terminates. Then we later show termination of the function for the desired set of states.

Second, the induction principle would be hard to manipulate in interactive proofs, even for a simple algorithm like *DFS*, let alone other more involved algorithms, as it would contain the entire algorithm and its control flow. The first step we perform is to create definitions corresponding to the different execution paths that each iteration could take. For each such execution path, we define 1. a predicate indicating that this path will be taken and 2. a function modelling the effect of the iteration on the state in this specific path. For *DFS*, we have four such execution paths, two of which are non-recursive. Below we show the auxiliary predicate indicating that the second recursive path will be taken and a function performing the same update that happens to the state when this execution path is taken. The other three predicates are called $\mathit{DFS\_cond}_1$, $\mathit{DFS\_ret\_cond}_1$, and $\mathit{DFS\_ret\_cond}_2$, and the update functions are called DFS_1, $\mathit{DFS\_ret}_1$, and $\mathit{DFS\_ret}_2$, for the first recursive call, and the two non-recursive calls, respectively.

$$\begin{array}{l}
\mathit{DFS\_cond}_2 :: (\mathit{'v},\ \mathit{'vset})\ \mathit{DFS\_state} \Rightarrow \mathit{bool} \\
\mathit{DFS\_cond}_2\ \mathit{dfs\_state} \\
= (\exists\, v\ \mathit{stack\_tl}. \\
\qquad v \neq t \wedge \mathcal{N}_G\ v -_G \mathit{seen}\ \mathit{dfs\_state} = \emptyset_V \wedge \\
\qquad \mathit{stack}\ \mathit{dfs\_state} = v\ \#\ \mathit{stack\_tl})
\end{array}$$

$$\begin{array}{l}
\mathit{DFS}_2 :: (\mathit{'v},\ \mathit{'vset})\ \mathit{DFS\_state} \Rightarrow (\mathit{'v},\ \mathit{'vset})\ \mathit{DFS\_state} \\
\mathit{DFS}_2\ \mathit{dfs\_state} = \mathit{dfs\_state}(\!|\mathit{stack} := \mathit{tl}\ (\mathit{stack}\ \mathit{dfs\_state})|\!)
\end{array}$$

We now prove the following theorem characterising *DFS*'s computation induction principle in terms of the auxiliary predicates and functions we defined.

$$\begin{array}{l}
\mathit{DFS\_dom}\ \mathit{dfs\_state} \wedge \\
(\forall\, \mathit{dfs\_state}. \\
\quad \mathit{DFS\_dom}\ \mathit{dfs\_state} \wedge \\
\quad (\mathit{DFS\_cond}_1\ \mathit{dfs\_state} \longrightarrow P\ (\mathit{DFS}_1\ \mathit{dfs\_state})) \wedge \\
\quad (\mathit{DFS\_cond}_2\ \mathit{dfs\_state} \longrightarrow P\ (\mathit{DFS}_2\ \mathit{dfs\_state})) \longrightarrow \\
\quad P\ \mathit{dfs\_state}) \longrightarrow \\
P\ \mathit{dfs\_state}
\end{array}$$

Note: the above induction principle is a streamlined version of an automatically generated computation induction principle. Also note that, since we did not prove that *DFS* terminates for all inputs, the induction principle is conditional: it applies to states satisfying a predicated *DFS_dom*, which is a predicate indicating that the function terminates for the given state.

24.1.4 Proving *DFS* correct

Now that we have modelled *DFS* and setup reasoning principles, we are ready to prove it correct. To do so, we devise a number of properties that, if true for a state, will hold for all states encountered throughout the execution of the algorithm, a.k.a. **loop invariants**. There are two main loop invariants. The first is the following:

```
invar_stack_walk :: ('v, 'vset) DFS_state ⇒ bool
invar_stack_walk dfs_state = vwalk [G]_G (rev (stack dfs_state))
```

That invariant implies that, if the algorithm terminates with success, the stack can be used to find a walk between the source and the destination.

The second invariant is the following:

```
invar_visited_through_seen :: ('v, 'vset) DFS_state ⇒ bool
invar_visited_through_seen dfs_state
= (∀v∈[seen dfs_state]_s.
      ∀p. vwalk_bet [G]_G v p t ∧ distinct p ⟶
           set p ∩ set (stack dfs_state) ≠ {})
```

That invariant implies that the target vertex is not reachable from the source if the algorithm finishes without success, i.e. if *return* (*DFS dfs_state*) = *NotReachable*.

To prove that either one of these is indeed an invariant, we use the induction principle we derived earlier. That means that, for each invariant, we have to consider the two recursive execution paths, leading to four proof obligations, two per invariant. To prove those obligations, however, we need the following further auxiliary invariants:

```
invar_well_formed :: ('v, 'vset) DFS_state ⇒ bool
invar_well_formed dfs_state = vset_inv (seen dfs_state)
```

$$
\begin{aligned}
&\textsf{invar\_seen\_stack} :: ({'}v,\ {'}vset)\ \mathit{DFS\_state} \Rightarrow \mathit{bool}\\
&\textsf{invar\_seen\_stack}\ \mathit{dfs\_state}\\
&= (\textsf{distinct}\ (\textsf{stack}\ \mathit{dfs\_state}) \wedge\\
&\quad \textsf{set}\ (\textsf{stack}\ \mathit{dfs\_state}) \subseteq [\textsf{seen}\ \mathit{dfs\_state}]_s \wedge\\
&\quad [\textsf{seen}\ \mathit{dfs\_state}]_s \subseteq \textsf{dVs}\ [G]_G)
\end{aligned}
$$

$$
\begin{aligned}
&\textsf{invar\_s\_in\_stack} :: ({'}v,\ {'}vset)\ \mathit{DFS\_state} \Rightarrow \mathit{bool}\\
&\textsf{invar\_s\_in\_stack}\ \mathit{dfs\_state}\\
&= (\textsf{stack}\ \mathit{dfs\_state} \neq [] \longrightarrow \textsf{last}\ (\textsf{stack}\ \mathit{dfs\_state}) = s)
\end{aligned}
$$

Naturally, each of these auxiliary invariants needs proving, increasing the number of proof obligations. We note that all proof obligations for all invariants, except one, which we discuss below, were automatically provable using standard automated proof tools, after setting them up to use results that we have proved about abstract graphs of the type $({'}v \times {'}v)\ \mathit{set}$. The only obligation that was not proved automatically is the following:

Lemma 24.1. $\textsf{DFS\_cond}_2\ \mathit{dfs\_state} \wedge \textsf{invar\_well\_formed}\ \mathit{dfs\_state} \wedge \textsf{invar\_seen\_stack}\ \mathit{dfs\_state} \wedge \textsf{invar\_visited\_through\_seen}\ \mathit{dfs\_state} \longrightarrow \textsf{invar\_visited\_through\_seen}\ (\textsf{DFS}_2\ \mathit{dfs\_state})$

Proof. Assume we have a walk p starting at v_1 and ending at t, and intersecting with the old stack $v_2\ \#\ \mathit{stack\_tl}$. We have to show that p intersects with $\mathit{stack\_tl}$. We have two cases:

- Case 1: If the point of intersection of the walk is in $\mathit{stack\_tl}$, then we are done.
- Case 2: If it intersects the old stack at v_2, which is the more interesting case as v_2 will not be in the new stack $\mathit{stack\_tl}$. First, this means that $p = p_1\ @\ [v_2]\ @\ p_2$, for some walks p_1 and p_2.
  Since the invariant holds for the old state, then $[v_2]\ @\ p_2$ intersects the old stack $v_2\ \#\ \mathit{stack\_tl}$. There are two cases which we need to consider here:
  - Case a: $p_2 = []$ This cannot be the case, since it would imply that $v_2 = t$ (recall that t is the target vertex), which violates the assumption of us being in the second recursive execution branch.
  - Case b: $p_2 \neq []$ From the current branch's assumptions, we know that $\textsf{hd}\ p_2$, which is a neighbour of v_2, is in $\textsf{seen}\ \mathit{dfs\_state}$. This means that, from the invariant at the current state $\mathit{dfs\_state}$, we can conclude that $v_2\ \#$

p_2 intersects with the old stack. However, since v_2 # p_2 is distinct, from *invar_seen_stack*, that means that p_2 cannot contain v_2. This means that p_2 intersects *stack_ tl*, which implies that p intersects with *stack_ tl*. This finishes our proof. □

After proving that the invariants hold, we have theorems of the following form:

Lemma 24.2. *DFS_dom dfs_ state* ∧ *invar_well_formed dfs_ state* ∧ *invar_seen_stack dfs_ state* ∧ *invar_visited_through_seen dfs_ state* ⟶ *invar_visited_through_seen* (*DFS dfs_ state*)

This theorem only states that, starting at a state for which we know *DFS* terminates and that the state satisfies the invariant, the state returned by *DFS* will also satisfy the invariant.

This leaves us with the task of showing that *DFS* terminates for all relevant program states. A standard method to show termination of recursive functions is by devising measure functions, i.e. functions mapping states to natural numbers, and showing that the value of the measure function decreases with every recursive call. An obvious measure function for *DFS* is the following:

```
call_1_measure :: ('v, 'vset) DFS_ state ⇒ nat
call_1_measure dfs_ state = card (dVs [G]_G − [seen dfs_ state]_s)
```

This measure function decreases the more vertices we have in the set of seen vertices. Note, however, that the value of this measure function only decreases in the first recursive execution branch; in the second recursive execution branch its value stays the same, as in that branch we only remove a vertex from the stack. We thus devise a second measure function for the second recursive execution branch:

```
call_2_measure :: ('v, 'vset) DFS_ state ⇒ nat
call_2_measure dfs_ state = card (set (stack dfs_ state))
```

Having more than one measure function somewhat complicates the termination proof, as we do not have one function that always decreases with recursive calls. A standard way to deal with that is by constructing a lexicographic ordering on the program states by combining different measure functions. This is specified as follows:

$\mathit{DFS\_term\_rel} :: (('v, 'vset)\ DFS\_state \times ('v, 'vset)\ DFS\_state)\ set$

$\mathit{DFS\_term\_rel} = \mathit{call\_1\_measure} <\!\!*mlex*\!\!> \mathit{call\_2\_measure} <\!\!*mlex*\!\!> \{\}$

This relation holds for two states dfs_state_1 and dfs_state_2 iff, either

- $\mathit{call\_1\_measure}\ dfs\_state_1 < \mathit{call\_1\_measure}\ dfs\_state_2$ or
- $\mathit{call\_1\_measure}\ dfs\_state_1 = \mathit{call\_1\_measure}\ dfs\_state_2$ and $\mathit{call\_2\_measure}\ dfs\_state_1 < \mathit{call\_2\_measure}\ dfs\_state_2$.

We show that, in both recursive calls, the resulting state is 'less than' the starting state w.r.t. this order.

$$\mathit{DFS\_cond_1}\ dfs\_state \wedge \mathit{invar\_well\_formed}\ dfs\_state \wedge \\ \mathit{invar\_seen\_stack}\ dfs\_state \longrightarrow \\ (\mathit{DFS_1}\ dfs\_state,\ dfs\_state) \in \mathit{DFS\_term\_rel}$$

$$\mathit{DFS\_cond_2}\ dfs\_state \wedge \mathit{invar\_well\_formed}\ dfs\_state \wedge \\ \mathit{invar\_seen\_stack}\ dfs\_state \longrightarrow \\ (\mathit{DFS_2}\ dfs\_state,\ dfs\_state) \in \mathit{DFS\_term\_rel}$$

Note the dependence on the fact that the starting state satisfies some of our invariants. This indicates that the algorithm only terminates for states satisfying those invariants. Indeed, we show that *DFS* terminates on any state satisfying those invariants:

$$\mathit{invar\_well\_formed}\ dfs\_state \wedge \mathit{invar\_seen\_stack}\ dfs\_state \longrightarrow \\ \mathit{DFS\_dom}\ dfs\_state$$

The last step here is to show that termination holds for an initial state satisfying those invariants. This state is defined as follows:

$\mathit{initial\_state} :: ('v, 'vset)\ DFS\_state$

$\mathit{initial\_state}$

$= (\!|\mathit{stack} = [s],\ \mathit{seen} = insert\ s\ \emptyset_V,\ \mathit{return} = \mathit{NotReachable}|\!)$

After showing that this state satisfies the invariants, which is trivial, we can finally show that *DFS* is correct.

Theorem 24.3. $\mathit{return}\ (DFS\ \mathit{initial\_state}) = \mathit{NotReachable} \longrightarrow$ $(\nexists p.\ \mathit{distinct}\ p \wedge \mathit{vwalk\_bet}\ [G]_G\ s\ p\ t)$

Theorem 24.4. $\mathit{return}\ (\mathit{DFS}\ \mathit{initial\_state}) = \mathit{Reachable} \longrightarrow$ $\mathit{vwalk\_bet}\ [G]_G\ s\ (\mathit{rev}\ (\mathit{stack}\ (\mathit{DFS}\ \mathit{initial\_state})))\ t$

We finally note that the correctness of the algorithm is proved, assuming that *DFS* axioms hold. This predicate summarises the assumptions we have on the implementations of the different ADTs we used and that the source is a vertex belonging to the graph in which we are searching. It is formally defined as follows:

$$\begin{aligned}&\mathit{DFS\_axioms}\\&= (\mathit{graph\_inv}\ G \wedge \mathit{finite}\ (\mathit{dom}\ (\mathit{lookup}\ G)) \wedge (\forall \mathit{vset}.\ \mathit{finite}\ [\mathit{vset}]_s) \wedge\\&\quad s \in \mathit{dVs}\ [G]_G)\end{aligned}$$

24.1.5 Executability

The final part of implementing and verifying an algorithm using our approach is making it executable by providing correct implementations to the *Pair_ Graph_ Specs* and the *Set2* ADTs. This is done using exactly the same approach discussed earlier in this book for providing implementations of the *Set* and *Map* ADTs, e.g. using red-black trees to implement sets of vertices and adjacency maps.

24.2 Breadth-First Search

Another standard way of traversing a graph is by traversing it **breadth-first**. Implementation-wise, this could be done by replacing the stack in DFS with a queue. Like DFS, there are many applications for breadth-first traversal, most notably, searching for a target vertex, i.e. breadth-first search (BFS). If one does that, in addition to the two guarantees we had for DFS (namely, DFS will find a walk iff there is one), we have the extra guarantee that there is not a shorter walk than the one found by BFS between the source and target.

24.2.1 Notions of Distance in a Directed Graph

As stated earlier, the main motivation for choosing a breadth-first traversal of a graph over a depth-first one is the guarantee it offers on the length of the returned walk, if there is such a walk. Here we formalise, for our abstract notion of directed graphs, notions that enable us to formally express properties related to walk-length optimality. The first such concept is the distance between two vertices:

$$\begin{aligned}&\mathit{d} :: ('v \times 'v)\ \mathit{set} \Rightarrow 'v \Rightarrow 'v \Rightarrow \mathit{enat}\\&\mathit{d}\ G\ u\ v = (\mathit{INF}\ p.\ \textbf{if}\ \mathit{vwalk\_bet}\ G\ u\ p\ v\ \textbf{then}\ \mathit{enat}\ (|p| - 1)\ \textbf{else}\ \infty)\end{aligned}$$

Above, $\mathsf{Inf}\ (\mathsf{range}\ f)$ could be read as $\text{argmin}_p f(p)$ in standard computer science literature, i.e. the p that minimises $f\ p$, for a function f. Note that the distance's value is of the type *enat*, which is constituted of all natural numbers and infinity (for a natural number x, $\mathsf{enat}\ x$ denotes the corresponding *enat*). The distance from a vertex u to v is considered to be infinite if there is not a walk from u to v.

The most important property of the concept of distance within a directed graph is that of the **triangle inequality**:

Theorem 24.5. $\mathsf{d}\ G\ u\ w \le \mathsf{d}\ G\ u\ v + \mathsf{d}\ G\ v\ w$

Another concept we need to define here is that of shortest walks.

$$\mathsf{shortest\_walk} :: ({}'v \times {}'v)\ set \Rightarrow {}'v \Rightarrow {}'v\ list \Rightarrow {}'v \Rightarrow bool$$

$$\mathsf{shortest\_walk}\ G\ u\ p\ v = (\mathsf{d}\ G\ u\ v = \mathsf{enat}\ (|p| - 1) \wedge \mathsf{vwalk\_bet}\ G\ u\ p\ v)$$

Finally, we define another notion of distances, whose use will become evident later on. This notion of distances is between a set of vertices and a vertex and is defined as follows:

$$\mathsf{D} :: ({}'v \times {}'v)\ set \Rightarrow {}'v\ set \Rightarrow {}'v \Rightarrow enat$$

$$\mathsf{D}\ G\ U\ v = (INF\ u{\in}U.\ \mathsf{d}\ G\ u\ v)$$

Intuitively, this is the distance between v and the closest member of U.

24.2.2 Modelling the Algorithm

In many applications (e.g. Aingworth et al. [1999]'s algorithm to bound graph diameters), one devises an algorithm that performs a breadth-first traversal and returns a **BFS-tree**. A BFS-tree is a subgraph of the directed graph, s.t. an edge is in the tree iff that edge was 'processed' during the breadth-first traversal of the directed graph under consideration. Figure 24.5 shows a directed graph and a BFS-tree[1] resulting from a BFS traversal. The important property that is needed in any application that uses BFS-trees is that the distance from the root to any vertex in the tree is equal to the distance between the two vertices in the traversed graph.

Below we model an algorithm that creates a slightly more general structure: a BFS directed acyclic graph (DAG). These are similar BFS-trees, but they can have multiple roots and are not forests, i.e. there could be more than one walk between a root and a vertex. These structures have applications in matching algorithms, e.g. Hopcroft and

[1] There could be more than one BFS-tree, depending on the non deterministic choice of neighbours to add to the queue.

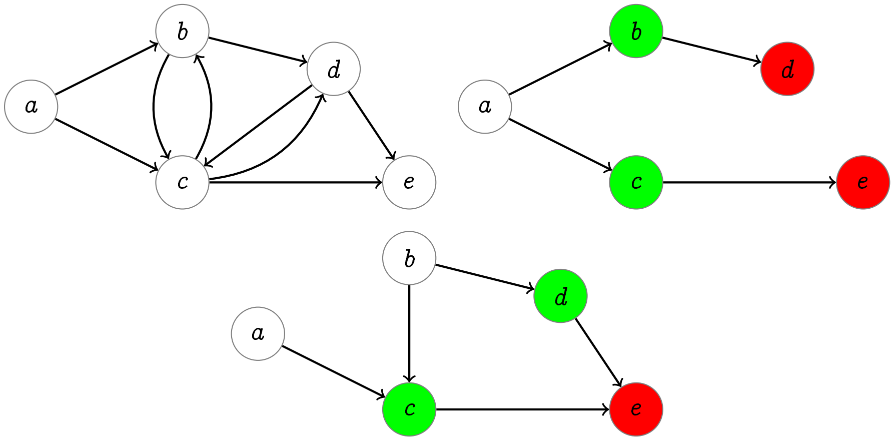


Figure 24.5 A directed graph, one of its BFS-trees rooted at vertex a, its BFS-DAG rooted at vertices a and b. In the latter two graphs, vertices are colour-coded (similar colours indicating similar distances) based on their distance from the root(s).

Karp [1973]'s algorithm for bipartite matching. Figure 24.5 shows a directed graph and a BFS-DAG resulting from a BFS traversal. A pseudo-code of the algorithm is shown below:

```
visited := current
while visited != empty do
  visited += current
  for each u in current do
    for each v in ((neighbourhood u) - visited) do
      parents += {(u,v)}
      current' += {v}
  current := current'
  current' := empty
```

The algorithm maintains variables modelled by the following state:

record (*'adjmap*, *'vset*) *BFS_state* = *current*::*'vset*
visited::*'vset*
DAG:: *'adjmap*

As the names of the elements of the state suggest: *current* is the set of vertices to be processed in the current iteration, *visited* is a set of vertices that were processed, and *DAG* is the BFS-DAG constructed by the algorithm.

To model this algorithm, we make a number of decisions demonstrating the process of modelling an algorithm for verification. The first such decision is, similar to *DFS*, that of using the existing ADTs for the needed operations. The next, and more relevant decision, is the level of detail at which we model the algorithm. A problem with modelling the algorithm as shown in the pseudo-code is that we have multiple nested loops. That would complicate the process of verifying the algorithm, as we would need multiple nested inductions, one per-iterative construct, to prove any fact about the algorithm. A way to avoid that is, instead of fully specifying the `for each` loop, we only assume a function that performs the computation expected from the `for each` loop. We then use the assumed properties of these functions to prove that the algorithm satisfies what is expected, if an implementation of these functions is provided. The way we do that is by using the interface for *BFS* in Figure 24.6. Based on that interface, we specify the algorithm as follows:

```
BFS :: ('adjmap, 'vset) BFS_state ⇒ ('adjmap, 'vset) BFS_state
BFS bfs_state
= (if current bfs_state ≠ ∅_V
   then let vis' = visited bfs_state ∪_G current bfs_state;
            par' = expand_tree (DAG bfs_state) (current bfs_state) vis';
            cur' = next_frontier (current bfs_state) vis'
        in BFS (bfs_state⦇parents := par', visited := vis', current := cur'⦈)
   else bfs_state)
```

We note that in addition to the applications of BFS-DAGs, here we consider an algorithm computing such DAGs as proving it correct requires some more involved graph-theoretic reasoning than that required in DFS or in a version of BFS that only computes a path between one source and one target. This helps deliver the main message of this chapter: demonstrating a methodology for the development of correct algorithms that need somewhat deep mathematical background/reasoning.

24.2.3 Proving *BFS* Correct

As discussed earlier, we have chosen to model *BFS* in a way that minimises complicated control flow. Thus, we do not have much complexity regarding the number of proof obligations we need to prove if we want to use the computation induction principle of *BFS*. Indeed, there is only one obligation, as there is only one recursive

ADT $BFS = Graph\colon Pair\_Graph\_Specs + set\_ops\colon Set2 +$

interface

$G :: {'}adjmap$

$srcs :: {'}vset$

$expand\_tree :: {'}adjmap \Rightarrow {'}vset \Rightarrow {'}vset \Rightarrow {'}adjmap$

$next\_frontier :: {'}vset \Rightarrow {'}vset \Rightarrow {'}vset$

specification

$\mathsf{graph\_inv}\ BFS\_tree \wedge vset\_inv\ frontier \wedge vset\_inv\ vis \wedge \mathsf{graph\_inv}\ G \longrightarrow$
$\mathsf{graph\_inv}\ (expand\_tree\ BFS\_tree\ frontier\ vis)$

$\mathsf{graph\_inv}\ BFS\_tree \wedge vset\_inv\ frontier \wedge vset\_inv\ vis \wedge \mathsf{graph\_inv}\ G \longrightarrow$
$[expand\_tree\ BFS\_tree\ frontier\ vis]_G$
$= [BFS\_tree]_G \cup$
$\quad \{(u,\ v) \mid u \in [frontier]_s \wedge v \in \mathsf{neighbourhood}\ [G]_G\ u - [vis]_s\}$

$vset\_inv\ frontier \wedge vset\_inv\ vis \wedge \mathsf{graph\_inv}\ G \longrightarrow$
$vset\_inv\ (next\_frontier\ frontier\ vis)$

$vset\_inv\ frontier \wedge vset\_inv\ vis \wedge \mathsf{graph\_inv}\ G \longrightarrow$
$[next\_frontier\ frontier\ vis]_s$
$= \bigcup\ \{\mathsf{neighbourhood}\ [G]_G\ u \mid u \in [frontier]_s\} - [vis]_s$

Figure 24.6 Interface of *BFS*. We omit the interface elements that come from either the ADTs *Pair_Graph_Specs* or *Set2*, as they are the same as the interface of *DFS*. The other elements of *BFS*'s interface are the input graph, the set of source vertices from which the traversal starts, and two functions *expand_tree* and *next_frontier* that are specified to compute what the `for each` loops are supposed to compute. The former function extends the BFS-DAG, and the latter one changes the current set of vertices being processed.

execution branch. However, the complexity here is mainly graph-theoretic. We need to show the following two properties for the computed BFS-DAG:

- The distance, in the BFS-DAG, between a root vertex of the BFS-DAG and any vertex that is not a root is the same as the distance between the two vertices in the original directed graph.
- Any walk in the BFS-DAG between a root vertex and another vertex is a shortest-walk in the BFS-DAG. Note that we need to show this property, as we are not computing a BFS-tree, i.e. we have no guarantee of uniqueness of walks.

Again, to show those two properties we first prove a number of loop-invariants. Those loop invariants are as follows:

$$
\begin{array}{l}
\textit{invar\_dist} :: ({'}\mathit{adjmap},\ {'}\mathit{vset})\ \mathit{BFS\_state} \Rightarrow \mathit{bool}\\
\textit{invar\_dist}\ \mathit{bfs\_state}\\
= (\forall v{\in}\textit{dVs}\ [G]_G\ -\ [\mathit{srcs}]_s.\\
\qquad v \in [\textit{visited}\ \mathit{bfs\_state}]_s \cup [\textit{current}\ \mathit{bfs\_state}]_s \longrightarrow\\
\qquad \textit{D}\ [G]_G\ [\mathit{srcs}]_s\ v = \textit{D}\ [\textit{DAG}\ \mathit{bfs\_state}]_G\ [\mathit{srcs}]_s\ v)
\end{array}
$$

$$
\begin{array}{l}
\textit{invar\_parents\_shortest\_paths} :: ({'}\mathit{adjmap},\ {'}\mathit{vset})\ \mathit{BFS\_state} \Rightarrow \mathit{bool}\\
\textit{invar\_parents\_shortest\_paths}\ \mathit{bfs\_state}\\
= (\forall u{\in}[\mathit{srcs}]_s.\\
\qquad \forall p\ v.\ \textit{vwalk\_bet}\ [\textit{DAG}\ \mathit{bfs\_state}]_G\ u\ p\ v \longrightarrow\\
\qquad\qquad \textit{enat}\ (|p| - 1) = \textit{D}\ [G]_G\ [\mathit{srcs}]_s\ v)
\end{array}
$$

$$
\begin{array}{l}
\textit{invar\_goes\_through\_current} :: ({'}\mathit{adjmap},\ {'}\mathit{vset})\ \mathit{BFS\_state} \Rightarrow \mathit{bool}\\
\textit{invar\_goes\_through\_current}\ \mathit{bfs\_state}\\
= (\forall u{\in}[\textit{visited}\ \mathit{bfs\_state}]_s \cup [\textit{current}\ \mathit{bfs\_state}]_s.\\
\qquad \forall v.\ v \notin [\textit{visited}\ \mathit{bfs\_state}]_s \cup [\textit{current}\ \mathit{bfs\_state}]_s \longrightarrow\\
\qquad\quad (\forall p.\ \textit{vwalk\_bet}\ [G]_G\ u\ p\ v \longrightarrow\\
\qquad\qquad \textit{set}\ p \cap [\textit{current}\ \mathit{bfs\_state}]_s \neq \{\}))
\end{array}
$$

Note that the last invariant is to make sure that, when the algorithm terminates, i.e. when *current* is empty, the BFS-DAG covers all vertices reachable from at least one of the roots.

Now, we give an overview of the proof that one of those three invariants holds, namely, $\mathsf{invar\_dist}$. We do so primarily to demonstrate abstract/graph-theoretic reasoning that is feasible using our approach of modelling graphs algorithmically and mathematically and the abstraction functions connecting the two representations.

Lemma 24.6. $\mathsf{BFS\_axiom} \wedge \mathsf{BFS\_cond}_1\ \mathit{bfs\_state} \wedge \mathsf{invar\_subsets}\ \mathit{bfs\_state} \wedge$
$\mathsf{invar\_well\_formed}\ \mathit{bfs\_state} \wedge \mathsf{invar\_dist\_bounded}\ \mathit{bfs\_state} \wedge$
$\mathsf{invar\_dist}\ \mathit{bfs\_state} \longrightarrow$
$\mathsf{invar\_dist}\ (\mathsf{BFS}_1\ \mathit{bfs\_state})$

Note that above $\mathsf{BFS\_cond}_1$ and BFS_1 are the auxiliary predicate and function characterising the only recursive execution branch of BFS.

Before we discuss the proof, we first note the auxiliary assumptions and invariants needed to show that this invariant is preserved. The first is an assumption stating the well-formedness of the graph which the algorithm processes; the second is an invariant ensuring that the well-formedness of the state is preserved; the third is an invariant stating important properties of the BFS-DAG and its relation to the input directed graph G.

$\mathsf{BFS\_axiom} :: \mathit{bool}$

$\mathsf{BFS\_axiom}$
$= (\mathsf{graph\_inv}\ G \wedge \mathsf{finite\_graph}\ G \wedge \mathsf{finite\_vsets} \wedge$
$\quad [\mathit{srcs}]_s \subseteq \mathsf{dVs}\ [G]_G \wedge$
$\quad (\forall u.\ \mathsf{finite}\ (\mathsf{neighbourhood}\ [G]_G\ u)) \wedge [\mathit{srcs}]_s \neq \{\} \wedge$
$\quad \mathit{vset\_inv}\ \mathit{srcs})$

$\mathsf{invar\_well\_formed} :: ('\mathit{adjmap}, '\mathit{vset})\ \mathit{BFS\_state} \Rightarrow \mathit{bool}$

$\mathsf{invar\_well\_formed}\ \mathit{bfs\_state}$
$= (\mathit{vset\_inv}\ (\mathsf{visited}\ \mathit{bfs\_state}) \wedge$
$\quad \mathit{vset\_inv}\ (\mathsf{current}\ \mathit{bfs\_state}) \wedge$
$\quad \mathsf{graph\_inv}\ (\mathsf{DAG}\ \mathit{bfs\_state}) \wedge \mathsf{finite}\ [\mathsf{current}\ \mathit{bfs\_state}]_s \wedge$
$\quad \mathsf{finite}\ [\mathsf{visited}\ \mathit{bfs\_state}]_s)$

$\mathsf{invar\_subsets} :: ('\mathit{adjmap}, '\mathit{vset})\ \mathit{BFS\_state} \Rightarrow \mathit{bool}$

$\mathsf{invar\_subsets}\ \mathit{bfs\_state}$
$= ([\mathsf{DAG}\ \mathit{bfs\_state}]_G \subseteq [G]_G \wedge [\mathsf{visited}\ \mathit{bfs\_state}]_s \subseteq \mathsf{dVs}\ [G]_G \wedge$

$$
\begin{aligned}
&[\mathsf{current}\ bfs\_state]_s \subseteq \mathsf{dVs}\ [G]_G\ \wedge\\
&\mathsf{dVs}\ [\mathsf{DAG}\ bfs\_state]_G \subseteq [\mathsf{visited}\ bfs\_state]_s \cup [\mathsf{current}\ bfs\_state]_s\ \wedge\\
&[srcs]_s \subseteq [\mathsf{visited}\ bfs\_state]_s \cup [\mathsf{current}\ bfs\_state]_s)
\end{aligned}
$$

Note: those two auxiliary invariants are proved independently of *invar_dist*, but we will not go into the details of those proofs.

Proof of Lemma 24.6. First, let $visited_0$ denote $\mathsf{visited}\ bfs\_state$, $visited_1$ denote $\mathsf{visited}\ (\mathsf{BFS}_1\ bfs\_state)$, DAG_0 denote $\mathsf{DAG}\ bfs\_state$, DAG_1 denote $\mathsf{DAG}\ (\mathsf{BFS}_1\ bfs\_state)$, $current_0$ denote $\mathsf{current}\ bfs\_state$, and $current_1$ denote $\mathsf{current}\ (\mathsf{BFS}_1\ bfs\_state)$.

To prove that invariant *invar_dist* holds, consider a vertex $v \in [visited_1]_s \cup [current_1]_s$. For this vertex, we need to show that $D\ [G]_G\ [srcs]_s\ v = D\ [DAG_1]_G\ [srcs]_s\ v$. Informally, we need to show that the distance from the sources to v in the input graph is the same as the distance in the BFS-DAG, after an iteration. We perform a case analysis.

- Case 1: $D\ [G]_G\ [srcs]_s\ v = \infty$, i.e. there is not a walk between any source and v. We know that $[DAG_1]_G \subseteq [G]_G$ from the invariant $\mathsf{invar\_subsets}\ bfs\_state$. The proof is finished by the following property of distances:

$$G \subseteq G' \longrightarrow \mathsf{D}\ G'\ Vs\ v \leq \mathsf{D}\ G\ Vs\ v \tag{24.1}$$

- Case 2: $D\ [G]_G\ [srcs]_s\ v \neq \infty$, i.e. there is a walk between some source u and v. Again, here we consider two further cases:
  - Case 2.a: $v \in [visited_0]_s \cup [current_0]_s$, i.e. v was already in the BFS-DAG before the current iteration starts. First, we have that $D\ [DAG_0]_G\ [srcs]_s\ v = D\ [G]_G\ [srcs]_s\ v$ because the invariant $\mathsf{invar\_dist}\ bfs\_state$ holds. We also have $D\ [DAG_0]_G\ [srcs]_s\ v = D\ [DAG_1]_G\ [srcs]_s\ v$ because 1. $[DAG_0]_G \subseteq [DAG_1]_G$ holds, implying that $D\ [DAG_1]_G\ [srcs]_s\ v \leq D\ [DAG_0]_G\ [srcs]_s\ v$, and 2. $D\ [DAG_0]_G\ [srcs]_s\ v = D\ [DAG_1]_G\ [srcs]_s\ v$, because $D\ [G]_G\ [srcs]_s\ v \leq D\ [DAG_1]_G\ [srcs]_s\ v$, using Inequality 24.1, and $D\ [DAG_1]_G\ [srcs]_s\ v \leq D\ [DAG_0]_G\ [srcs]_s\ v$, also using Inequality 24.1.
  - Case 2.b: $v \notin [visited_0]_s \cup [current_0]_s$, i.e. v has been added to the BFS-DAG during the current iteration. Since $v \in [visited_1]_s \cup [current_1]_s$, there must exist v', s.t. $v \in \mathsf{neighbourhood}\ [G]_G\ v'$ and $v' \in [current_0]_s$. First,

note that we have that

$$
\begin{aligned}
D\ [G]_G\ [\mathit{srcs}]_s\ v &= D\ [G]_G\ [\mathit{srcs}]_s\ v' + 1 && (24.2)\\
&= D\ [\mathit{DAG}_0]_G\ [\mathit{srcs}]_s\ v' + 1 && (24.3)\\
&= D\ [\mathit{DAG}_1]_G\ [\mathit{srcs}]_s\ v' + 1 && (24.4)
\end{aligned}
$$

We now prove the theorem by contradiction, i.e. by assuming $D\ [G]_G\ [\mathit{srcs}]_s\ v \neq D\ [\mathit{DAG}_1]_G\ [\mathit{srcs}]_s\ v$. From this assumption, since $[\mathit{DAG}_1]_G \subseteq [G]_G$, and from Inequality 24.1, we have that $D\ [G]_G\ [\mathit{srcs}]_s\ v < D\ [\mathit{DAG}_1]_G\ [\mathit{srcs}]_s\ v$. From this and the above three equations, we have that $D\ [\mathit{DAG}_1]_G\ [\mathit{srcs}]_s\ v' + 1 < D\ [\mathit{DAG}_1]_G\ [\mathit{srcs}]_s\ v$. This leaves us with a contradiction since $v \in \mathsf{neighbourhood}\ [G]_G\ v'$, since $v' \in [\mathit{current}_0]_s$ and from the assumption of this case, i.e. $v \notin [\mathit{visited}_0]_s \cup [\mathit{current}_0]_s$, which means that v was added to $\mathit{DAG}1$ in this iteration. We now prove the three equations from above, to finish the proof.

Equation 24.2 is the most involved here. To see why it holds, we refute the two cases which violate it. First, $D\ [G]_G\ [\mathit{srcs}]_s\ v' + 1 < D\ [G]_G\ [\mathit{srcs}]_s\ v$ cannot hold, as that would violate the triangle inequality. Second, consider the case when $D\ [G]_G\ [\mathit{srcs}]_s\ v < D\ [G]_G\ [\mathit{srcs}]_s\ v' + 1$ holds. Deriving a contradiction here depends on assuming that invariant *invar_dist_bounded* holds. The definition of this invariant is as follows:

$$
\begin{aligned}
&\mathsf{invar\_dist\_bounded} :: ('\mathit{adjmap}, '\mathit{vset})\ \mathit{BFS\_state} \Rightarrow \mathit{bool}\\
&\mathsf{invar\_dist\_bounded}\ \mathit{bfs\_state}\\
&= (\forall v{\in}[\mathsf{visited}\ \mathit{bfs\_state}]_s \cup [\mathsf{current}\ \mathit{bfs\_state}]_s.\\
&\qquad \forall u.\ \mathsf{D}\ [G]_G\ [\mathit{srcs}]_s\ u \leq \mathsf{D}\ [G]_G\ [\mathit{srcs}]_s\ v \longrightarrow\\
&\qquad\qquad u \in [\mathsf{visited}\ \mathit{bfs\_state}]_s \cup [\mathsf{current}\ \mathit{bfs\_state}]_s)
\end{aligned}
$$

The contradiction follows from the assumption of this case (i.e. Case 2.b) and the fact that $v' \in [\mathit{current}_0]_s$.

Equation 24.3 holds because this invariant that we are proving holds in the initial state, i.e. *invar_dist* $\mathit{bfs\_state}$, and since $v' \in [\mathit{current}_0]_s$.

Equation 24.4 holds because we have 1. $D\ [\mathit{DAG}_1]_G\ v' \leq D\ [\mathit{DAG}_0]_G\ v'$, since $[\mathit{DAG}_0]_G \subseteq [\mathit{DAG}_1]_G$ holds by construction, 2. $D\ [G]_G\ v' \leq D\ [\mathit{DAG}_1]_G\ v'$, since $[\mathit{DAG}_1]_G \subseteq [G]_G$ holds from *invar_subsets*, and, lastly, 3. $D\ [G]_G\ v' \leq D\ [\mathit{DAG}_0]_G\ v'$, since *invar_dist* $\mathit{bfs\_state}$ holds by assumption, and since $v' \in [\mathit{current}_0]_s$. □

We note a number of points regarding this proof. First, in addition to the three main invariants, we had to show four further auxiliary invariants, e.g. *invar_dist_bounded*.

For the majority of those invariants, the arguments were about distances and deriving contradictions from different properties of distances. The only exception was proving the invariant *invar_goes_through_current*, where the argument was mainly about properties of walks. We will not discuss the details of those proofs here. Some of the more involved properties of distances we used other than the triangle inequality include the following:

$$\mathsf{D}\ G\ U\ v \neq \infty \wedge u \in U \wedge \mathsf{d}\ G\ u\ v = \mathsf{D}\ G\ U\ v \longrightarrow \\ (\exists p.\ \mathsf{shortest\_walk}\ G\ u\ (u \# p)\ v \wedge \mathsf{set}\ p \cap U = \{\})$$

$$\mathsf{D}\ G\ U\ v = \mathsf{d}\ G\ u\ v \wedge u \in U \wedge \mathsf{shortest\_walk}\ G\ u\ p\ v \wedge w \in \mathsf{set}\ p \longrightarrow \\ \mathsf{D}\ G\ U\ w = \mathsf{d}\ G\ u\ w$$

$$\mathsf{D}\ G\ U\ v = \mathsf{d}\ G\ u\ v \wedge u \in U \wedge \mathsf{shortest\_walk}\ G\ u\ (p_1\ @\ w \# p_2)\ v \wedge \\ w \in V \wedge (\forall v' \in V.\ \mathsf{D}\ G\ U\ v' = \mathsf{d}\ G\ u\ w) \longrightarrow \\ \mathsf{D}\ G\ (U \cup V)\ v = \mathsf{D}\ G\ U\ v - \mathsf{d}\ G\ u\ w$$

Deriving these properties for distances is not immediately straightforward. However, the fact that we proved them on the abstract representation of directed graphs made deriving them much easier compared to proving them directly on graphs as represented by *Pair_ Graph_ Specs*. Although our algorithm was defined in terms of *Pair_ Graph_ Specs* as a graph model, the proofs were made easier by the abstraction functions connecting *Pair_ Graph_ Specs* and the abstract mathematical representation; and our configuring basic proof automation to translate goals automatically using the abstraction functions.

For termination, we used the following measure functions and lexicographic ordering:

$$\mathsf{call\_1\_measure\_1} :: ('adjmap,\ 'vset)\ BFS\_state \Rightarrow nat$$
$$\mathsf{call\_1\_measure\_1}\ bfs\_state \\ = \mathsf{card}\ (\mathsf{dVs}\ [G]_G - ([\mathsf{visited}\ bfs\_state]_s \cup [\mathsf{current}\ bfs\_state]_s))$$

$$\mathsf{call\_1\_measure\_2} :: ('adjmap,\ 'vset)\ BFS\_state \Rightarrow nat$$
$$\mathsf{call\_1\_measure\_2}\ bfs\_state = \mathsf{card}\ [\mathsf{current}\ bfs\_state]_s$$

$$\mathsf{BFS\_term\_rel} :: \\ (('adjmap,\ 'vset)\ BFS\_state \times ('adjmap,\ 'vset)\ BFS\_state)\ set$$

$\mathit{BFS\_term\_rel}$
$= \mathit{call\_1\_measure\_1} <\!*mlex*\!> \mathit{call\_1\_measure\_2} <\!*mlex*\!> \{\}$

The main intuition here is that in all iterations, except the last one, we visit more vertices, thus decreasing the first measure function. In the last iteration, we visit no more vertices, but empty the set *current*.

The initial state, which we prove is terminating and for which we have the final correctness theorems is the following:

$\mathit{initial\_state} :: ('adjmap,\ 'vset)\ \mathit{BFS\_state}$
$\mathit{initial\_state} = (\!|\mathit{parents} = \emptyset_G,\ \mathit{current} = \mathit{srcs},\ \mathit{visited} = \emptyset_V|\!)$

Finally, the main three properties we show for the algorithm are as follows:

Theorem 24.7. $\mathit{BFS\_axiom} \wedge u \in [\mathit{srcs}]_s \wedge t \notin [\mathit{visited}\ (\mathit{BFS}\ \mathit{initial\_state})]_s \longrightarrow (\nexists p.\ \mathit{vwalk\_bet}\ [G]_G\ u\ p\ t)$

Theorem 24.8. $\mathit{BFS\_axiom} \wedge t \in [\mathit{visited}\ (\mathit{BFS}\ \mathit{initial\_state})]_s - [\mathit{srcs}]_s \longrightarrow D\ [G]_G\ [\mathit{srcs}]_s\ t = D\ [\mathit{DAG}\ (\mathit{BFS}\ \mathit{initial\_state})]_G\ [\mathit{srcs}]_s\ t$

Theorem 24.9. $\mathit{BFS\_axiom} \wedge u \in [\mathit{srcs}]_s \wedge$
$\mathit{vwalk\_bet}\ [\mathit{DAG}\ (\mathit{BFS}\ \mathit{initial\_state})]_G\ u\ p\ v \longrightarrow$
$\mathit{enat}\ (|p| - 1) = D\ [G]_G\ [\mathit{srcs}]_s\ v$

24.3 Chapter Notes

Our representation of directed graphs does not allow for singleton vertices in the graph, i.e. any vertex in the graph is connected to another vertex via an edge. Another alternative is to represent the graph as a pair (V, E), with a set of vertices and a set of edges. There is a complication with this: one has to make sure that all the graph's edges are incident only to its vertices. There is also the representation of graphs by Noschinski [Noschinski 2015]. This representation is more abstract than the one we use here, but has not been tested in substantial algorithmic developments. Other representations of graphs have been investigated in the course of other verification efforts of graph algorithms [Lammich and Nipkow 2019, Lammich and Sefidgar 2019].

Our way of modelling iterative algorithms has the main advantage that it requires little extra machinery than basic specification of recursive functions. There are other ways of modelling iterative algorithms, most notably using whilecombinators [Berghofer and Nipkow 2002] or monads [Lammich and Tuerk 2012]. The use of these other methods is primarily geared towards enabling more automatic proofs using program logics, like Hoare logic or separation logic. Such automation is most useful for reasoning at the level of the data structures, pointers, or program implementation more generally. The methodology we used here is largely manual, and it pays off if the primary effort in proving the algorithm correct is of an abstract mathematical nature, rather on program or data structure specific constructs, e.g. if reasoning about mathematical properties of concepts like matchings [Abdulaziz et al. 2019] or flows [Lammich and Sefidgar 2019] constitutes most proof effort.

Our proofs of correctness of DFS and BFS are performed at a relatively abstract mathematical level. This is in comparison to other expositions [Cormen et al. 2009], where correctness proofs are performed on full implementations, where the behaviour of data structures is not abstracted away. In our case, this is enabled by 1. using ADTs to specify the data structures, 2. devising a background theory on directed graphs that is suited for conducting proofs at a mathematical level, and 3. connecting the ADTs to the background library on directed graphs using abstraction functions, allowing us to state all specifications and conduct almost all proofs in terms of the abstract graph library.

In essence, the approach we followed is one implementation of step-wise refinement [Wirth 1971], where our graph abstraction lemmas and theorem prover automation can be seen as basic data refinement infrastructure. There are other more involved approaches to implement **step-wise refinement** within a theorem prover. One such implementation is by Lammich [Lammich 2019]. In his approach, one would start with an abstract mathematical description of an algorithm, prove it correct, and then derive a more concrete version, and prove their equivalence. His approach emphasises custom automation techniques and the use of separation logic to provide imperative implementations of the ADTs, which can be much faster in practice than the purely functional implementations of ADTs discussed here. However, this comes at a cost of low-level tinkering of automation and usually pays off when the main goal is a high-performance piece of verified software. Another approach is that taken by Greenaway et al. [Greenaway et al. 2012], where one would start with a C-language implementation, and tooling is provided to parse the programs as well as derive equivalent abstract mathematical functions, and automatically proving the data refinement relations.

Another interesting approach is that of lifting and transfer [Huffman and Kuncar 2013]. That approach implements **parametric reasoning** as first noted by Wadler [Wadler 1989]. There the focus is on showing an equivalence between two types and then using that equivalence to derive theorems about one type from corresponding theorems on the other type. This method has the advantage of making the automation connecting the two representations of the graphs more principled than general purpose theorem proving methods, which we use here.

25 Fast String Search by Knuth–Morris–Pratt

Lawrence C. Paulson

Nothing could be simpler than searching for occurrences of a string in a text file, yet we have two sophisticated algorithms for doing this: one by Knuth, Morris and Pratt (KMP), the other by Boyer and Moore. Both were published in 1977, when 1 MB was thought to be a lot of memory. Nowadays strings can be orders of magnitude longer, making the need for efficiency all the greater. Bioinformatics requires searching truly gigantic strings: of nucleotides (when working with genomes) and amino acids (in the case of proteins). Here we look at KMP, the simpler of the two algorithms.

The naive algorithm aligns the pattern p with the text string a, comparing corresponding characters from left to right, and in case of a mismatch, shifting one position along a and starting again. This is actually fine under plausible assumptions. The alphabet surely has more than one character, and if furthermore the characters in the string are random then the expected length of a partial match will be finite, since it involves the sum of a geometric series. Ergo, linear time.

But if the text is not random then the worst-case time is $O(mn)$, where m and n are the lengths of p and a. For suppose that p and a both have the form `xxx...xy`, consisting entirely of the letter `x` except having a single `y` at the end. The naive algorithm will make m comparisons, failing at the last one; then it will shift p one position along a even though there is no hope of a match. This wasteful search will continue until a is exhausted.

The idea of KMP is to exploit the knowledge gained from the partial match, never re-comparing characters that matched. At the first mismatched character, it shifts p as far to the right as is safely possible. To do so, it consults a precomputed table, based on the pattern p, identifying repeated substrings for which the current, failed partial match could become the first part of a full match.

In the case of our example, the successful match of the first part of the pattern, namely `x...x`, means we already know the previous $m-1$ characters of a, so instead of shifting one position along and checking p from the beginning, we can check from where we left off, i.e. its penultimate character. The search will still fail until the final y is reached, but without any superfluous comparisons. The algorithm takes $\Theta(m+n)$ time, where the $\Theta(m)$ part comes from the pre-computation of the table.

25.1 Preliminaries: Difference Arrays

Our task is to take an imperative algorithm designed nearly half a century ago and express it in a functional style, retaining the possibility of efficient execution. Strictly speaking, there are two algorithms: the computation of the table, and the string search using the table. Neither would normally be seen as functional, but both algorithms are simple **while** loops, easily expressed as tail-recursive functions. Arrays are used, and random access is necessary. However, in the building phase, the table entries are added one after another, and the search does no array updates at all.

Because the original algorithms are imperative, their use of arrays is **single-threaded**. That means there is a single thread of updates starting from the initial value to the final array. It implies that updates can be done without copying: the previous array value can safely be destroyed. This conception can be realised by an ordinary array as supported by the hardware, augmented with a difference structure to deal with any array accesses that are not single-threaded. Provided there are none of those, performance can be good.

This data structure is called a **difference array**, and is part of the Collections framework [Lammich 2009]. This chapter uses the following notation for array operations:

- A !! n to look up an array element (indexed from 0)
- $A[n ::= x]$ to update an array
- $\|A\|$ for the number of elements
- *array* x n to create an n-element array, all elements filled with x.

All but the last of these is assumed to take constant time.

25.2 Matches between Strings

A key concept is that of an n-character **match** between two strings a and b, starting at positions i and j, respectively (indexed from 0).

matches :: $'a\ array \Rightarrow nat \Rightarrow 'a\ array \Rightarrow nat \Rightarrow nat \Rightarrow bool$

matches $a\ i\ b\ j\ n$
$= (i + n \le \|a\| \wedge j + n \le \|b\| \wedge (\forall k{<}n.\ a\ !!\ (i + k) = b\ !!\ (j + k)))$

```
x y z x y z x z x y
      x y z x y z x z x y
            x y z x y z x z x y
                    x y z x y z x z x y
```

Figure 25.1 Identifying prefixes in the search pattern

Most of its properties are obvious. It always holds when $n = 0$, provided i and j lie within the range of their respective strings. A simple but valuable fact is **weakening** to get a shorter match: if *matches* $a\ i\ b\ j\ n$ and $k \leq n$ then

$$\textit{matches}\ a\ i\ b\ j\ k \quad \text{and} \quad \textit{matches}\ a\ (i + k)\ b\ (j + k)\ (n - k).$$

Sometimes we look for matches between the pattern p with the text a, but when building the table we will be matching prefixes of p with other sections of p.

25.3 The Next-Match Table

As noted above, the table identifies repetitions in the pattern that open the possibility that the current failed match may yet form part of a successful match. For example, suppose our search pattern p is `xyzxyzxzxy`. And suppose we have matched `xyz`$\underline{\texttt{x}}$ in the string followed by a mismatch. The point is that the final `x` could be the start of an occurrence of p in the string. Similarly, if we have matched `xyz`$\underline{\texttt{xy}}$, `xyz`$\underline{\texttt{xyz}}$ or `xyz`$\underline{\texttt{xyzx}}$, the underlined section is a partial match of p and the search for a full match should continue from that point. But if we match `xyzxyzxz`, no suffix of this matches a prefix of p. Finally, matching `xyzxyzxz`$\underline{\texttt{x}}$ let us use the final x as the start of a match. (Matching the whole of p would leave `xy` as the start of another possible match, but the algorithm below stops after the first.) Figure 25.1 illustrates the situation.

The corresponding next-match table is

x	y	z	x	y	z	x	z	x	y
0	0	0	0	1	2	3	4	0	1

These numbers are indices into p, numbering from 0. So for example 4 above tells us that at the position shown, we have successfully matched the first four characters of p and should start comparing at $p[4]$, which is `y`.

Now we are ready for the following predicate, which defines the next available match following a failed comparison:

$$\mathit{is\_next} :: {}'a\ \mathit{array} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat} \Rightarrow \mathit{bool}$$

$$\begin{array}{l} \mathit{is\_next}\ p\ j\ n \\ = (n < j \wedge \mathit{matches}\ p\ (j - n)\ p\ 0\ n \wedge \\ \quad (\forall m.\ n < m < j \longrightarrow \neg\ \mathit{matches}\ p\ (j - m)\ p\ 0\ m)) \end{array}$$

In other words, n is the largest possible that is less than j and with an n-character match of a prefix of p with a substring of p ending at j.

The following two lemmas capture the essence of this. First, if the first j characters of the pattern already match (ending at position i in the text), and n is the next match, then indeed the first n characters of p match the text (again ending at i).

Lemma 25.1. $\mathit{matches}\ a\ (i - n)\ p\ 0\ n$, **provided**

- $\mathit{matches}\ a\ (i - j)\ p\ 0\ j$
- $\mathit{is\_next}\ p\ j\ n$
- $j \leq i$

Proof. We have $\mathit{matches}\ a\ (i - n)\ p\ (j - n)\ n$ by weakening the given assumption. Moreover, we have $\mathit{matches}\ p\ (j - n)\ p\ 0\ n$ by the definition of $\mathit{is\_next}$. The conclusion is immediate by transitivity. □

The second lemma considers the same situation (a j-character match ending at i) and tells us that the "next match", n, is really maximal: there does not exist a full match of p ending at k for any k, where $i - j < k < i - n$.

Lemma 25.2. $\neg\ \mathit{matches}\ a\ k\ p\ 0\ \|p\|$, **provided**

- $\mathit{matches}\ a\ (i - j)\ p\ 0\ j$
- $\mathit{is\_next}\ p\ j\ n$
- $j \leq i$
- $i - j < k < i - n$

Proof. Let m denote $i - k$. Then $\neg\ \mathit{matches}\ a\ (i - m)\ p\ 0\ m$ by the definition of $\mathit{is\_next}$ and weakening. Further weakening using $m < \|p\|$ yields the desired $\neg\ \mathit{matches}\ a\ (i - m)\ p\ 0\ \|p\|$. □

Therefore, using the next-match table to shift the pattern along will give us a partial match, which we can hope to complete, safe in the knowledge that there are no matches starting in the skipped-over region. All we have to do is build this table.

25.4 Building the Table: Loop Body and Invariants

Although this is a book of functional algorithms, here we basically have a **while** loop. Maintaining $j < i \le \|p\|$, it builds a match of the first j characters of p with a substring of p ending at i, meanwhile filling the next table *nxt* with the corresponding j values. At a mismatch, it consults its own table—exactly as the main string search will do—for the longest possible match that still holds. In the imperative pseudo-code, m denotes $\|p\|$, the length of p.

```
nxt[1] := 0; i := 1; j := 0;
while i < m-1 do
  if p[i] = p[j] then
    begin i := i+1; j := j+1; nxt[i] := j end
  else
    if j = 0 then begin i := i+1; nxt[i] := 0 end
    else j := nxt[j]
```

The loop body, expressed as a function, takes the pattern p and the three loop variables *nxt*, i, j:

$$\begin{aligned}
&\textit{buildtab\_step} :: \\
&\quad {}'a\ \mathit{array} \Rightarrow \mathit{nat\ array} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat\ array} \times \mathit{nat} \times \mathit{nat} \\
&\textit{buildtab\_step}\ p\ \mathit{nxt}\ i\ j \\
&= (\mathbf{if}\ p\ !!\ i = p\ !!\ j\ \mathbf{then}\ (\mathit{nxt}[i+1 ::= j+1],\ i+1,\ j+1) \\
&\qquad \mathbf{else\ if}\ j = 0\ \mathbf{then}\ (\mathit{nxt}[i+1 ::= 0],\ i+1,\ j)\ \mathbf{else}\ (\mathit{nxt},\ i,\ \mathit{nxt}\ !!\ j))
\end{aligned}$$

To verify the **while** loop requires defining the **loop invariant**: a property of the loop variables that holds initially and is preserved in each iteration.

$$\begin{aligned}
&\textit{buildtab\_invariant} :: {}'a\ \mathit{array} \Rightarrow \mathit{nat\ array} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat} \Rightarrow \mathit{bool} \\
&\textit{buildtab\_invariant}\ p\ \mathit{nxt}\ i\ j \\
&= (\|\mathit{nxt}\| = \|p\| \wedge i \le \|p\| \wedge j < i \wedge \textit{matches}\ p\ (i - j)\ p\ 0\ j\ \wedge \\
&\qquad (\forall k.\ 0 < k \le i \longrightarrow \textit{is\_next}\ p\ k\ (\mathit{nxt}\ !!\ k)) \wedge \\
&\qquad (\forall k.\ j + 1 < k < i + 1 \longrightarrow \neg\ \textit{matches}\ p\ (i + 1 - k)\ p\ 0\ k))
\end{aligned}$$

It's natural to regard this as the conjunction of six simpler invariants, some of which obviously hold, but some are nontrivial and depend on one another. The length of *nxt* obviously doesn't change, and since $i + 1 < \|p\|$ holds prior to execution of the loop body, $i \leq \|p\|$ holds and this inequality could even be strict. As for $j < i$, the critical case is when $p \,!!\, i \neq p \,!!\, j$ and $j > 0$; the point is that $nxt \,!!\, j < j$ by the definition of *is_next* and the corresponding invariant. The invariant that we have a match of length j has the same critical case and holds for the same reason.

We are left with two nontrivial invariants, and must prove they are preserved by every execution of the loop body.

- That the next-match table is indeed built correctly (up to i)
- That there cannot exist a match of length $> j{+}1$ starting earlier in p than the match we have.

Lemma 25.3. *is_next* $p\ k\ (nxt' \,!!\, k)$, **provided**

- $(nxt', i', j') =$ *buildtab_step* $p\ nxt\ i\ j$
- *buildtab_invariant* $p\ nxt\ i\ j$
- $i + 1 < \|p\|$
- $0 < k \leq i'$

Proof. Consider *buildtab_step* $p\ nxt\ i\ j$. If $p \,!!\, i = p \,!!\, j$ then $i' = i + 1$ and $j' = j + 1$; then *matches* $p\ (i - j)\ p\ 0\ (j + 1)$ using the *matches* part of the invariant, hence *is_next* $p\ (i + 1)\ (j + 1)$ by definition and the prior invariant. Therefore, the updated table, $nxt' = nxt[i + 1 ::= j + 1]$, satisfies the conclusion.

So we can assume $p \,!!\, i \neq p \,!!\, j$. If $j = 0$ then $i' = i + 1$. The character clash implies $\neg$ *matches* $p\ (i - j)\ p\ 0\ (j + 1)$ and therefore *is_next* $p\ (i + 1)\ 0$, validating the updated next-match table, $nxt' = nxt[i + 1 ::= 0]$. In the final case, when $j > 0$, both i and nxt are left unchanged, making the conclusion trivial. □

Lemma 25.4. $\neg$ *matches* $p\ (i' + 1 - k)\ p\ 0\ k$, **provided**

- $(nxt', i', j') =$ *buildtab_step* $p\ nxt\ i\ j$
- *buildtab_invariant* $p\ nxt\ i\ j$
- $\|p\| \geq 2$
- $i + 1 < \|p\|$
- $j' + 1 < k < i' + 1$

Proof. Consider *buildtab_step* $p\ nxt\ i\ j$. If $p \,!!\, i = p \,!!\, j$ then $i' = i + 1$ and $j' = j + 1$; the conclusion follows from the same invariant for i and j. So we can assume $p \,!!\, i \neq p \,!!\, j$. If $j = 0$ then we need to show

$$\neg\ \textit{matches}\ p\ (i + 2 - k)\ p\ 0\ k\ \ \textbf{if}\ \ 1 < k\ \textbf{and}\ k < i + 2.$$

If $k = 2$ then $i + 2 - k = i$ and we know $p \,!!\, i \neq p \,!!\, 0$, so *matches* $p\ i\ p\ 0\ k$ is false; otherwise it follows by instantiating the same invariant with $k - 1$.

The remaining case is when $p \mathbin{!!} i \neq p \mathbin{!!} j$ and $j > 0$. Then $i' = i$ and $j' = nxt \mathbin{!!} j$, so we need to show

$$\neg \textsf{\textit{matches}}\ p\ (i + 1 - k)\ p\ 0\ k\ \ \textbf{if}\ \ nxt \mathbin{!!} j + 1 < k\ \textbf{and}\ k < i + 2.$$

This is trivial if $k > j + 1$ because the invariant holds beforehand, and if $k = j + 1$ because $p \mathbin{!!} i \neq p \mathbin{!!} j$. So we can assume $k \leq j$ and assume for contradiction that the match holds. Write $k' = k - 1$. Then we have

$\neg$ *matches* $p\ (j - k')\ p\ 0\ k'$, by the invariant *is_next* $p\ j\ (nxt \mathbin{!!} j)$
matches $p\ (j - k')\ p\ (i - k')\ k'$, by the invariant *matches* $p\ 0\ p\ (i - j)\ j$
matches $p\ (i - k')\ p\ 0\ k'$, weakening the negated conclusion

The desired contradiction follows by the transitivity of *matches*. □

To summarize: we have proved that *buildtab_invariant* is preserved by *buildtab*:

Corollary 25.5. *buildtab_invariant* $p\ nxt'\ i'\ j'$, **provided**

- $(nxt', i', j') =$ *buildtab_step* $p\ nxt\ i\ j$
- *buildtab_invariant* $p\ nxt\ i\ j$
- $i + 1 < \|p\|$

25.5 Building the Table: Outer Loop

Now that we know that the loop body preserves the invariant, we are ready to define the actual function to build the next-match table. The loop itself is the obvious recursion:

```
buildtab :: 'a array ⇒ nat array ⇒ nat ⇒ nat ⇒ nat array
buildtab p nxt i j
= (if i + 1 < ||p||
   then let (nxt', i', j') = buildtab_step p nxt i j
        in buildtab p nxt' i' j'
   else nxt)
```

The key correctness property of the constructed table is not hard to prove. We must assume that the invariant holds initially.

Lemma 25.6. *is_next* $p\ k$ (*buildtab* $p\ nxt\ i\ j \mathbin{!!} k$), **provided**

- *buildtab_invariant* $p\ nxt\ i\ j$
- $0 < k < \|p\|$

Proof by computation induction on *buildtab*. If $i + 1 < \|p\|$, *buildtab_step* yields (nxt', i', j') also satisfying the invariant (by Corollary 25.5) and by IH the result of the recursive call has the desired *is_next* property. Conversely, if not $i + 1 < \|p\|$, the invariant implies the desired property of nxt. □

It is convenient to define a top-level function to call *buildtab*. It starts the loop with appropriate initial values, which can trivially be shown to establish the invariant, and catches a degenerate case to return a null table when p is trivial.

$$\textsf{\textit{table}} :: {}'a\ array \Rightarrow nat\ array$$

$$\textsf{\textit{table}}\ p = (\textbf{\textsf{if}}\ 1 < \|p\|\ \textbf{\textsf{then}}\ \textsf{\textit{buildtab}}\ p\ (\textsf{\textit{array}}\ 0\ \|p\|)\ 1\ 0\ \textbf{\textsf{else}}\ \textsf{\textit{array}}\ 0\ \|p\|)$$

By Lemma 25.6 we have all we need to know about the table-building function:

$$0 < j < \|p\| \longrightarrow \textsf{\textit{is\_next}}\ p\ j\ (\textsf{\textit{table}}\ p\ !!\ j) \tag{25.1}$$

25.6 Building the Table: Termination

It turns out that *buildtab* does not terminate on all inputs. For example, if $i = 0$, $j = 1$, $\|p\| > 1$, $p\ !!\ i \neq p\ !!\ j$, $p\ !!\ j = j$, then *buildtab_step* $p\ nxt\ i\ j = (nxt, i, j)$ and thus *buildtab* loops. We have not encountered non-termination before in this book and it raises two fundamental questions: is computation induction valid and can we even define *buildtab* in a logic of total functions, which HOL is.

Luckily, *buildtab* terminates on all inputs that satisfy the invariant: At every recursive call, either

- i increases by 1, with j unchanged or increased by 1, or
- i stays unchanged while j is replaced by $nxt\ !!\ j$, and $nxt\ !!\ j < j$ by the invariant.

In each of these cases, the integer quantity $2 \cdot \|p\| - 2 \cdot i + j$ decreases, and it is nonnegative because $i \leq \|p\|$ by the invariant. Therefore, execution terminates, and the number of calls to *buildtab_step* is linear in $\|p\|$. Since each step—a couple of comparisons and a couple of assignments—clearly takes constant time, the overall running time is linear.

The proof of termination justifies the use of computation induction whenever we can assume that the invariant holds initially.

Defining functions that need non terminate is a subtle issue in a logic of total functions like HOL. Luckily, *buildtab* is tail-recursive (which is not a coincidence: every **while** loop corresponds to a tail-recursive function). That fact allows us to define *buildtab* without having to prove termination: it is consistent to assume the existence of f satisfying $f(x) = f(x+1)$, since any constant function will do, unlike the apparently similar $f(x) = f(x+1)+1$.

We conclude this section with a formal counterpart of the above informal linear running time argument by means of a running time function for *buildtab*. Ironically, the very difficulty of *buildtab*'s termination proof complicates this step. Time functions are defined by equations of the form $T_f\ p = \mathcal{T}[\![e]\!] + 1$, which are not tail-recursive (if f occurs in e). For example, $f\ (C\,x) = f\,x$ induces $T_f\ (C\,x) = T_f\ x + 1$. However, we can easily turn T_f into a tail-recursive function with an accumulating time parameter: $T_f\ (C\ x)\ t = T_f\ x\ (t + 1)$. This leads to the following definition of $T_{buildtab}$:

```
T_buildtab :: 'a array ⇒ nat array ⇒ nat ⇒ nat ⇒ nat ⇒ nat
T_buildtab p nxt i j t
= (if i + 1 < ||p||
   then let (nxt', i', j') = buildtab_step p nxt i j
        in T_buildtab p nxt' i' j' (t + 1)
   else t)
```

The following result is proved similarly to Lemma 25.6.

Lemma 25.7. *buildtab_invariant* $p\ nxt\ i\ j \longrightarrow$
$T_{buildtab}\ p\ nxt\ i\ j\ t \leq 2 \cdot \|p\| - 2 \cdot i + j + t$

Plugging in the initial values, we find that

$$2 \leq \|p\| \longrightarrow T_{buildtab}\ p\ (\mathit{array}\ 0\ \|p\|)\ 1\ 0\ 0 \leq 2 \cdot (\|p\| - 1)$$

The precondition $2 \leq \|p\|$ is required because *buildtab_invariant* holds initially only in that case: $2 \leq \|p\| \longrightarrow$ *buildtab_invariant* p (*array* $0\ \|p\|$) $1\ 0$

The summary so far: we can build the next-match table, and in linear time. Now we are ready to search.

25.7 KMP String Search: Loop Body and Invariants

Like last time, let's begin with a **while** loop and then analyse the corresponding functional version. In this pseudocode, m and n denote the lengths of p and a, respectively. It closely resembles the previous algorithm, except it doesn't build a table, and it compares p with a rather than with itself.

```
i := 0; j := 0; nxt := table(p);
while j<m and i<n do
  if a[i] = p[j] then
    begin i := i+1; j := j+1 end
  else
    if j = 0 then i := i+1
    else j := nxt[j];
if j=m then i-m else i
```

The last line returns the result of the algorithm: if $j = m$, the whole pattern has been matched and $i - m$ is the beginning of the (first) occurrence of the pattern; otherwise i will be n, an indication that the pattern has not been found.

In the loop body, only i and j are modified, but the string, the pattern and the next-match table also need to be available. Hence the functional version takes all of them as arguments, but returns only the new values of i and j:

$$
\begin{aligned}
&\textsf{KMP\_step} :: {}'a\ \mathit{array} \Rightarrow \mathit{nat}\ \mathit{array} \Rightarrow {}'a\ \mathit{array} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat} \times \mathit{nat}\\
&\textsf{KMP\_step}\ p\ \mathit{nxt}\ a\ i\ j\\
&= (\textbf{if}\ a\ !!\ i = p\ !!\ j\ \textbf{then}\ (i + 1, j + 1)\\
&\quad\ \textbf{else if}\ j = 0\ \textbf{then}\ (i + 1, 0)\ \textbf{else}\ (i, \mathit{nxt}\ !!\ j))
\end{aligned}
$$

Once again, we need an invariant relating these quantities, which must be preserved at every loop iteration. This invariant is simpler because the tough intellectual work has been done already. It asserts that there is a match between the first j characters of p and the text, ending at i; moreover, there is no match of the whole of p with the text prior to that point.

$$
\begin{aligned}
&\textsf{KMP\_invariant} :: {}'a\ \mathit{array} \Rightarrow {}'a\ \mathit{array} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat} \Rightarrow \mathit{bool}\\
&\textsf{KMP\_invariant}\ p\ a\ i\ j\\
&= (j \le \|p\| \land j \le i \land i \le \|a\| \land \textsf{matches}\ a\ (i - j)\ p\ 0\ j\ \land\\
&\quad\ (\forall k{<}i - j.\ \neg\ \textsf{matches}\ a\ k\ p\ 0\ \|p\|))
\end{aligned}
$$

This property is preserved in each step provided $j < \|p\|$ and $i < \|a\|$. If $a\ !!\ i = p\ !!\ j$, or if $j = 0$, then the conclusion is trivial. The only interesting case is when $a\ !!\ i \neq p\ !!\ j$ and $j > 0$. Then we need to show the existence of a match of length $\mathit{nxt}\ !!\ j$, but that is immediate by the already established correctness of the next-match table. Finally, we need to show $\neg$ *matches* $a\ k\ p\ 0\ \|p\|$ for $k < i - \mathit{nxt}\ !!\ j$. We know that $k \neq i - j$ by the mismatch that just occurred, so either $k < i - j$, when the result is immediate by the given invariant, or $k > i - j$, when the result holds by Lemma 25.2.

25.8 KMP String Search: Outer Loop

Like last time, we express the **while** loop using recursion. The two active loop variables are i and j, but the function takes additional arguments m, n and nxt to prevent their being re-computed at every iteration. Their values will be $\|p\|$, $\|a\|$, and *table* p, respectively.

```
search ::
nat ⇒ nat ⇒ nat array ⇒ 'a array ⇒ 'a array ⇒ nat ⇒ nat ⇒ nat × nat
search m n nxt p a i j
= (if j < m ∧ i < n
    then let (i', j') = KMP_step p nxt a i j in search m n nxt p a i' j'
    else (i, j))
```

The following function is the “top level” version, invoking the search loop with appropriate initial values. That includes building the table, and the loop invariant is established vacuously.

```
KMP_search :: 'a array ⇒ 'a array ⇒ nat × nat
KMP_search p a = search ||p|| ||a|| (table p) p a 0 0
```

Note that the definition of *search* raises the same termination problems we already faced with *buildtab*. Termination again requires $nxt \,!!\, j < j$. This time it follows from the correctness of *table* (25.1) if we know $nxt =$ *table* p.

25.9 KMP String Search: Correctness

The following predicate expresses the correctness of the result (as computed in the last line of the imperative algorithm). There are two possibilities. Termination before the end of the text string is reached ($r < \|a\|$) signifies success. Conversely, $r = \|a\|$ implies failure.

```
first_occur :: 'a array ⇒ 'a array ⇒ nat ⇒ bool
first_occur p a r
= ((r < ||a|| ⟶ matches a r p 0 ||p||) ∧ (∀k<r. ¬ matches a k p 0 ||p||))
```

Lemma 25.8. *first_occur* $p\ a$ (**if** $j' = \|p\|$ **then** $i' - \|p\|$ **else** i'), **provided**

- $(i', j') =$ *search* $\|p\|\ \|a\|$ (*table* p) $p\ a\ i\ j$
- *KMP_invariant* $p\ a\ i\ j$

Proof by computation induction on *search*. We have $j \leq m$ and $i \leq n$ by the invariant. If $j < m$ and $i < n$ then we obtain the result by IH (because *KMP_step* preserves the invariant). Conversely, if $j = m$ or $i = n$ then the success or failure, respectively, follows by the invariant. □

As a corollary we obtain correctness of *KMP_search* because *KMP_search* establishes *KMP_invariant*.

Corollary 25.9. $(i, j) =$ *KMP_search* $p\ a \longrightarrow$
first_occur $p\ a$ (**if** $j = \|p\|$ **then** $i - \|p\|$ **else** i)

The proof of linearity of *search* is almost identical to that of Lemma 25.6, except that the quantity that decreases is $2 \cdot \|a\| - 2 \cdot i + j$, which is nonnegative because $i \leq \|a\|$. Its initial value is $2 \cdot \|a\|$ because those of i and j are both zero. So the loop body can execute at most $2 \cdot \|a\|$ times. It's not hard to see that this worst possible outcome occurs with the pathological string search mentioned at the beginning of this chapter. Even so, it is linear.

Chapter Notes

Acknowledgement. This development closely follows a formal verification of the Knuth–Morris–Pratt algorithm by Jean-Christophe Filliâtre using Why3. Due to the need for high performance in the era of gigabyte memories, innumerable variations exist. This version already achieves linear worst-case performance, and exhibits a pleasing symmetry between the table-building and search algorithms.

The original paper on KMP [Knuth et al. 1977], seemingly written by Knuth himself, is extremely clear. The realities of computing in the 1970s are evident in his suggestion that the string being searched might be held on an external file and that the naive search algorithm could introduce buffering issues, since after every failure of a match the algorithm would go back and rescan characters possibly no longer in main memory.

26 Huffman's Algorithm

Jasmin Blanchette

Huffman's algorithm [Huffman 1952] is a simple and elegant procedure for constructing a binary tree with minimum weighted path length—a measure of cost that considers both the lengths of the paths from the root to the leaf nodes and the weights associated with the leaf nodes. The algorithm's main application is data compression: by equating leaf nodes with characters and weights with character frequencies, we can use it to derive optimum binary codes. A **binary code** is a map from characters to nonempty sequences of bits.

This chapter presents Huffman's algorithm and its optimality proof. In a slight departure from the rest of this book, the emphasis is more on graphical intuitions and less on rigorous logical arguments.

26.1 Binary Codes

Suppose we want to encode strings over a finite source alphabet as sequences of bits. Fixed-length codes such as ASCII are simple and fast, but they generally waste space. If we know the frequency w_a of each source symbol a, we can save space by using shorter code words for the most frequent symbols. We say that a variable-length code is **optimum** if it minimizes the sum $\sum_a w_a \delta_a$, where δ_a is the length of the binary code word for a.

As an example, consider the string abacabad. Encoding it with the code

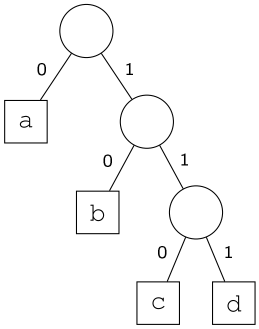
$$C_1 = \{\mathrm{a} \mapsto 0, \mathrm{b} \mapsto 10, \mathrm{c} \mapsto 110, \mathrm{d} \mapsto 111\}$$

gives the 14-bit code word 01001100100111. The code C_1 is optimum: no code that unambiguously encodes source symbols one at a time could do better than C_1 on the input abacabad. With a fixed-length code such as

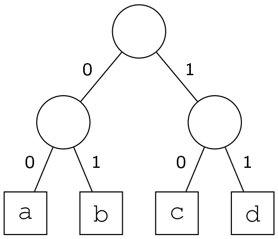
$$C_2 = \{\mathrm{a} \mapsto 00, \mathrm{b} \mapsto 01, \mathrm{c} \mapsto 10, \mathrm{d} \mapsto 11\}$$

we need at least 16 bits to encode the same string.

Binary codes can be represented by binary trees. For example, the trees



and



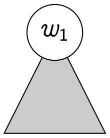
correspond to C_1 and C_2. The code word for a given symbol can be obtained as follows: start at the root and descend toward the leaf node associated with the symbol one node at a time. Emit a 0 whenever the left child of the current node is chosen and a 1 whenever the right child is chosen. The generated sequence of 0s and 1s is the code word.

To avoid ambiguities, we require that only leaf nodes are labeled with symbols. This ensures that no code word is a prefix of another. Moreover, it is sufficient to consider only full binary trees (trees whose inner nodes all have two children), because any node with only one child can advantageously be eliminated by removing it and letting the child take its parent's place.

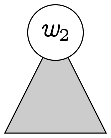
Each node in a code tree is assigned a **weight**. For a leaf node, the weight is the frequency of its symbol; for an inner node, it is the sum of the weights of its subtrees. In diagrams, we often annotate the nodes with their weights.

26.2 The Algorithm

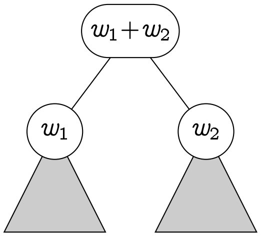
Huffman's algorithm is a very simple procedure for constructing an optimum code tree for specified symbol frequencies. It works as follows: first, create a list of leaf nodes, one for each symbol in the alphabet, taking the given symbol frequencies as node weights. The nodes must be sorted in increasing order of weight. Second, pick the two trees



and



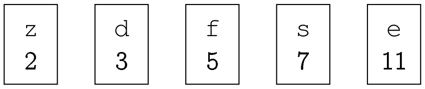
with the lowest weights and insert the tree



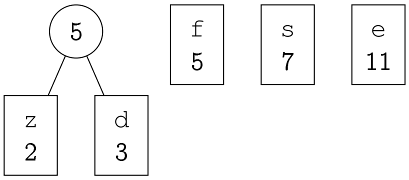
into the list so as to keep it ordered. Finally, repeat the process until only one tree is left in the list.

As an illustration, executing the algorithm for the frequencies $f_{\mathtt{d}} = 3$, $f_{\mathtt{e}} = 11$, $f_{\mathtt{f}} = 5$, $f_{\mathtt{s}} = 7$, and $f_{\mathtt{z}} = 2$ gives rise to the following sequence of states:

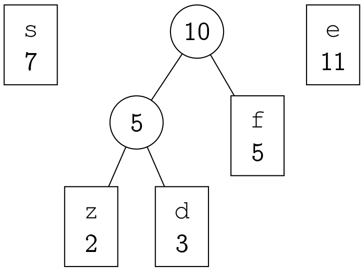
1.



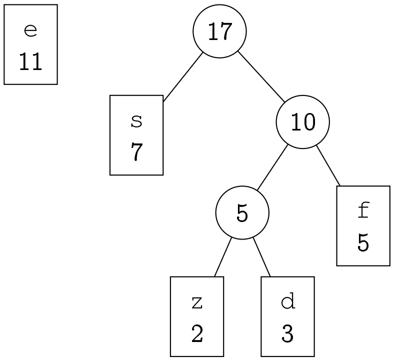
2.



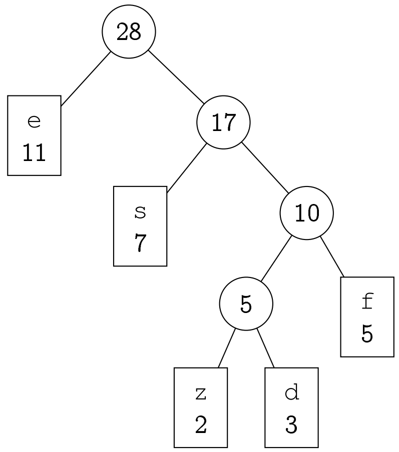
3.



4.



5.



The resulting tree is optimum for the given frequencies.

26.3 The Implementation

The functional implementation of the algorithm relies on the following type:

```
datatype 'a tree = Leaf nat 'a | Node nat ('a tree) ('a tree)
```

Leaf nodes are of the form *Leaf* w a, where a is a symbol and w is the frequency associated with a, and inner nodes are of the form *Node* w t_1 t_2, where t_1 and t_2 are the left and right subtrees and w caches the sum of the weights of t_1 and t_2. The *cachedWeight* function extracts the weight stored in a node:

```
cachedWeight :: 'a tree ⇒ nat
cachedWeight (Leaf w _) = w
cachedWeight (Node w _ _) = w
```

The implementation builds on two additional auxiliary functions. The first one, *uniteTrees*, combines two trees by adding an inner node above them:

```
uniteTrees :: 'a tree ⇒ 'a tree ⇒ 'a tree
uniteTrees t1 t2 = Node (cachedWeight t1 + cachedWeight t2) t1 t2
```

The second function, *insortTree*, inserts a tree into a list sorted by cached weight, preserving the sort order:

$$
\begin{aligned}
&\mathit{insortTree} :: {}'a\ \mathit{tree} \Rightarrow {}'a\ \mathit{tree\ list} \Rightarrow {}'a\ \mathit{tree\ list}\\
&\mathit{insortTree}\ u\ [] = [u]\\
&\mathit{insortTree}\ u\ (t\ \#\ ts)\\
&= (\mathbf{if}\ \mathit{cachedWeight}\ u \le \mathit{cachedWeight}\ t\ \mathbf{then}\ u\ \#\ t\ \#\ ts\\
&\quad \mathbf{else}\ t\ \#\ \mathit{insortTree}\ u\ ts)
\end{aligned}
$$

The main function that implements Huffman's algorithm follows:

$$
\begin{aligned}
&\mathit{huffman} :: {}'a\ \mathit{tree\ list} \Rightarrow {}'a\ \mathit{tree}\\
&\mathit{huffman}\ [t] = t\\
&\mathit{huffman}\ (t_1\ \#\ t_2\ \#\ ts) = \mathit{huffman}\ (\mathit{insortTree}\ (\mathit{uniteTrees}\ t_1\ t_2)\ ts)
\end{aligned}
$$

The function should initially be invoked with a nonempty list of leaf nodes sorted by weight. It repeatedly unites the first two trees of the list it receives as argument until a single tree is left.

26.4 Basic Auxiliary Functions Needed for the Proof

This section introduces basic concepts such as alphabet, consistency, and optimality, which are needed to state the correctness and optimality of Huffman's algorithm. The next section introduces more specialized functions that arise in the proof.

The *alphabet* of a code tree is the set of symbols appearing in the tree's leaf nodes:

$$
\begin{aligned}
&\mathit{alphabet} :: {}'a\ \mathit{tree} \Rightarrow {}'a\ \mathit{set}\\
&\mathit{alphabet}\ (\mathit{Leaf}\ \_\ a) = \{a\}\\
&\mathit{alphabet}\ (\mathit{Node}\ \_\ t_1\ t_2) = \mathit{alphabet}\ t_1 \cup \mathit{alphabet}\ t_2
\end{aligned}
$$

A tree is *consistent* if for each inner node the alphabets of the two subtrees are disjoint. Intuitively, this means that a symbol occurs in at most one leaf node. Consistency is a sufficient condition for δ_a (the length of the code word for a) to be uniquely defined. This well-formedness property appears as an assumption in many of the lemmas. The definition follows:

```
consistent :: 'a tree ⇒ bool
consistent (Leaf _ _) = True
consistent (Node _ t1 t2)
= (alphabet t1 ∩ alphabet t2 = {} ∧ consistent t1 ∧ consistent t2)
```

The *depth* of a symbol (which we wrote as δ_a above) is the length of the path from the root to that symbol, or equivalently the length of the code word for the symbol:

```
depth :: 'a tree ⇒ 'a ⇒ nat
depth (Leaf _ _) _ = 0
depth (Node _ t1 t2) a
= (if a ∈ alphabet t1 then depth t1 a + 1
   else if a ∈ alphabet t2 then depth t2 a + 1 else 0)
```

By convention, symbols that do not occur in the tree or that occur at the root of a one-node tree are given a depth of 0. If a symbol occurs in several leaf nodes (of an inconsistent tree), the depth is arbitrarily defined in terms of the leftmost node labeled with that symbol.

The *height* of a tree is the length of the longest path from the root to a leaf node, or equivalently the length of the longest code word:

```
height :: 'a tree ⇒ nat
height (Leaf _ _) = 0
height (Node _ t1 t2) = max (height t1) (height t2) + 1
```

The **frequency** of a symbol (which we wrote as w_a above) is the sum of the weights attached to the leaf nodes labeled with that symbol:

```
freq :: 'a tree ⇒ 'a ⇒ nat
freq (Leaf w a) b = (if b = a then w else 0)
freq (Node _ t1 t2) b = freq t1 b + freq t2 b
```

For consistent trees, the sum comprises at most one nonzero term. The frequency is then the weight of the leaf node labeled with the symbol, or 0 if there is no such node.

Two trees are **comparable** if they have the same alphabet and symbol frequencies. This is an important concept because it allows us to state not only that the tree constructed by Huffman's algorithm is optimum but also that it has the expected alphabet and frequencies.

The *weight* function returns the weight of a tree:

$$\begin{aligned}
&\mathit{weight} :: {}'a\ \mathit{tree} \Rightarrow \mathit{nat}\\
&\mathit{weight}\ (\mathit{Leaf}\ w\ \_) = w\\
&\mathit{weight}\ (\mathit{Node}\ \_\ t_1\ t_2) = \mathit{weight}\ t_1 + \mathit{weight}\ t_2
\end{aligned}$$

In the *Node* case, we ignore the weight cached in the node and instead compute the tree's weight recursively.

The *cost* (or **weighted path length**) of a consistent tree is the sum

$$\sum_{a \in \mathit{alphabet}\ t} \mathit{freq}\ t\ a \cdot \mathit{depth}\ t\ a$$

which we wrote as $\sum_a w_a \delta_a$ above. It is defined recursively by

$$\begin{aligned}
&\mathit{cost} :: {}'a\ \mathit{tree} \Rightarrow \mathit{nat}\\
&\mathit{cost}\ (\mathit{Leaf}\ \_\ \_) = 0\\
&\mathit{cost}\ (\mathit{Node}\ \_\ t_1\ t_2) = \mathit{weight}\ t_1 + \mathit{cost}\ t_1 + \mathit{weight}\ t_2 + \mathit{cost}\ t_2
\end{aligned}$$

A tree is *optimum* iff its cost is not greater than that of any comparable tree:

$$\begin{aligned}
&\mathit{optimum} :: {}'a\ \mathit{tree} \Rightarrow \mathit{bool}\\
&\mathit{optimum}\ t\\
&= (\forall u.\ \mathit{consistent}\ u \wedge \mathit{alphabet}\ t = \mathit{alphabet}\ u \wedge \mathit{freq}\ t = \mathit{freq}\ u \longrightarrow\\
&\qquad\quad \mathit{cost}\ t \le \mathit{cost}\ u)
\end{aligned}$$

Tree functions are readily generalized to lists of trees, or **forests**. For example, the alphabet of a forest is defined as the union of the alphabets of its trees. The forest generalizations have a subscript '$_F$' attached to their name (e.g., $\mathit{alphabet}_F$).

26.5 Other Functions Needed for the Proof

The optimality proof needs to interchange nodes in trees, to replace a two-leaf subtree with weights w_1 and w_2 by a single leaf node of weight $w_1 + w_2$ and vice versa, and to refer to the two symbols with the lowest frequencies. These concepts are represented by seven functions: *swapSyms*, *swapLeaves*, *swapFourSyms*, *mergeSibling*, *sibling*, *splitLeaf*, and *minima*.

The interchange function *swapSyms* takes a tree t and two symbols a, b, and exchanges the symbols:

```
swapSyms :: 'a tree ⇒ 'a ⇒ 'a ⇒ 'a tree
swapSyms t a b = swapLeaves t (freq t a) a (freq t b) b
```

The definition relies on the following auxiliary function:

```
swapLeaves :: 'a tree ⇒ nat ⇒ 'a ⇒ nat ⇒ 'a ⇒ 'a tree
swapLeaves (Leaf w_c c) w_a a w_b b
= (if c = a then Leaf w_b b else if c = b then Leaf w_a a else Leaf w_c c)
swapLeaves (Node w t_1 t_2) w_a a w_b b
= Node w (swapLeaves t_1 w_a a w_b b) (swapLeaves t_2 w_a a w_b b)
```

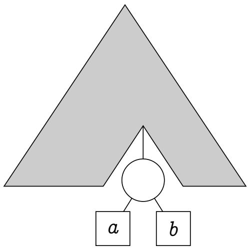
The following lemma captures the intuition that to minimize the cost more frequent symbols should be encoded using fewer bits than less frequent ones:

Lemma 26.1. $\mathit{consistent}\ t \wedge a \in \mathit{alphabet}\ t \wedge b \in \mathit{alphabet}\ t \wedge$
$\mathit{freq}\ t\ a \leq \mathit{freq}\ t\ b \wedge \mathit{depth}\ t\ a \leq \mathit{depth}\ t\ b \longrightarrow$
$\mathit{cost}\ (\mathit{swapSyms}\ t\ a\ b) \leq \mathit{cost}\ t$

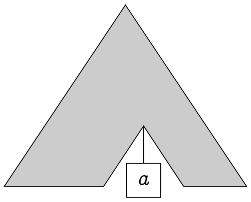
The four-way symbol interchange function *swapFourSyms* takes four symbols a, b, c, d with $a \neq b$ and $c \neq d$, and exchanges them so that a and b occupy c's and d's positions. A naive definition of this function would be $\mathit{swapSyms}\ (\mathit{swapSyms}\ t\ a\ c)\ b\ d$. This naive definition fails in the face of aliasing: if $a = d$, but $b \neq c$, then $\mathit{swapFourSyms}\ a\ b\ c\ d$ would wrongly leave a in b's position. Instead, we use this definition:

```
swapFourSyms :: 'a tree ⇒ 'a ⇒ 'a ⇒ 'a ⇒ 'a ⇒ 'a tree
swapFourSyms t a b c d
= (if a = d then swapSyms t b c
   else if b = c then swapSyms t a d
        else swapSyms (swapSyms t a c) b d)
```

Given a symbol a, the *mergeSibling* function transforms the tree



into



The frequency of a in the resulting tree is the sum of the original frequencies of a and b. The function is defined by the equations

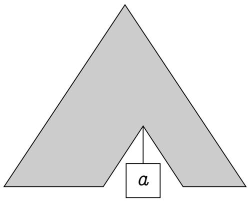
```
mergeSibling :: 'a tree ⇒ 'a ⇒ 'a tree
mergeSibling (Leaf w_b b) _ = Leaf w_b b
mergeSibling (Node w (Leaf w_b b) (Leaf w_c c)) a
= (if a = b ∨ a = c then Leaf (w_b + w_c) a
   else Node w (Leaf w_b b) (Leaf w_c c))
mergeSibling (Node w (Node v va vb) t_2) a
= Node w (mergeSibling (Node v va vb) a) (mergeSibling t_2 a)
mergeSibling (Node w t_1 (Node v va vb)) a
= Node w (mergeSibling t_1 a) (mergeSibling (Node v va vb) a)
```

The *sibling* function returns the label of the node that is the (left or right) sibling of the node labeled with the given symbol a in tree t. If a is not in t's alphabet or it occurs in a node with no sibling leaf node, we simply return a. This gives us the nice property that if t is consistent, then *sibling* $t\ a \neq a$ if and only if a has a sibling. The definition, which is omitted here, distinguishes the same cases as *mergeSibling*.

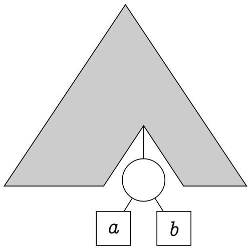
Using the *sibling* function, we can state that merging two sibling leaf nodes with weights w_a and w_b decreases the cost by $w_a + w_b$:

Lemma 26.2. *consistent* $t \wedge$ *sibling* $t\ a \neq a \longrightarrow$
cost (*mergeSibling* $t\ a$) + *freq* $t\ a$ + *freq* t (*sibling* $t\ a$) = *cost* t

The *splitLeaf* function undoes the merging performed by *mergeSibling*: given two symbols a, b and two frequencies w_a, w_b, it transforms



into



In the resulting tree, a has frequency w_a and b has frequency w_b. We normally invoke *splitLeaf* with w_a and w_b such that $\mathit{freq}\ t\ a = w_a + w_b$. The definition follows:

$$
\begin{aligned}
&\mathit{splitLeaf} :: \ 'a\ \mathit{tree} \Rightarrow \mathit{nat} \Rightarrow\ 'a \Rightarrow \mathit{nat} \Rightarrow\ 'a \Rightarrow\ 'a\ \mathit{tree}\\
&\mathit{splitLeaf}\ (\mathit{Leaf}\ w_c\ c)\ w_a\ a\ w_b\ b\\
&= (\mathbf{if}\ c = a\ \mathbf{then}\ \mathit{Node}\ w_c\ (\mathit{Leaf}\ w_a\ a)\ (\mathit{Leaf}\ w_b\ b)\ \mathbf{else}\ \mathit{Leaf}\ w_c\ c)\\
&\mathit{splitLeaf}\ (\mathit{Node}\ w\ t_1\ t_2)\ w_a\ a\ w_b\ b\\
&= \mathit{Node}\ w\ (\mathit{splitLeaf}\ t_1\ w_a\ a\ w_b\ b)\ (\mathit{splitLeaf}\ t_2\ w_a\ a\ w_b\ b)
\end{aligned}
$$

Splitting a leaf node with weight $w_a + w_b$ into two sibling leaf nodes with weights w_a and w_b increases the cost by $w_a + w_b$:

Lemma 26.3. $\mathit{consistent}\ t \wedge a \in \mathit{alphabet}\ t \wedge \mathit{freq}\ t\ a = w_a + w_b \longrightarrow \mathit{cost}\ (\mathit{splitLeaf}\ t\ w_a\ a\ w_b\ b) = \mathit{cost}\ t + w_a + w_b$

Finally, the *minima* predicate expresses that two symbols a, b have the lowest frequencies in the tree t and that $\mathit{freq}\ t\ a \le \mathit{freq}\ t\ b$:

$$
\begin{aligned}
&\mathit{minima} :: \ 'a\ \mathit{tree} \Rightarrow\ 'a \Rightarrow\ 'a \Rightarrow \mathit{bool}\\
&\mathit{minima}\ t\ a\ b\\
&= (a \in \mathit{alphabet}\ t \wedge b \in \mathit{alphabet}\ t \wedge a \neq b \wedge\\
&\quad (\forall c{\in}\mathit{alphabet}\ t.\\
&\qquad c \neq a \longrightarrow c \neq b \longrightarrow \mathit{freq}\ t\ a \le \mathit{freq}\ t\ c \wedge \mathit{freq}\ t\ b \le \mathit{freq}\ t\ c))
\end{aligned}
$$

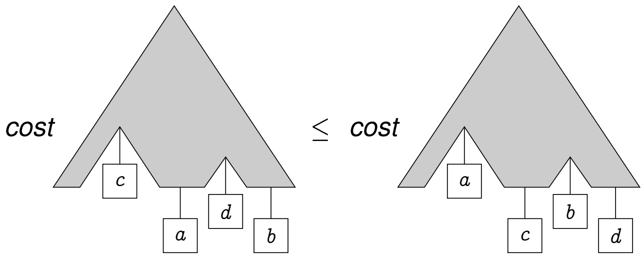
26.6 The Key Lemmas and Theorems

It is easy to prove that the tree returned by Huffman's algorithm preserves the alphabet, consistency, and symbol frequencies of the original forest:

$$
\begin{aligned}
&ts \neq [] \longrightarrow \mathit{alphabet}\ (\mathit{huffman}\ ts) = \mathit{alphabet}_F\ ts\\
&\mathit{consistent}_F\ ts \wedge ts \neq [] \longrightarrow \mathit{consistent}\ (\mathit{huffman}\ ts)\\
&ts \neq [] \longrightarrow \mathit{freq}\ (\mathit{huffman}\ ts)\ a = \mathit{freq}_F\ ts\ a
\end{aligned}
$$

The main difficulty is to prove the optimality of the tree constructed by Huffman's algorithm. We need to introduce three lemmas before we can present the optimality theorem.

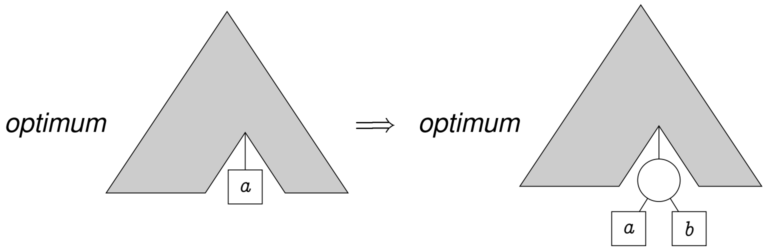
First, if a and b are minima and c and d are at the very bottom of the tree, then exchanging a and b with c and d does not increase the tree's cost. Graphically, we have



Lemma 26.4. $\mathit{consistent}\ t \wedge \mathit{minima}\ t\ a\ b \wedge$
$c \in \mathit{alphabet}\ t \wedge d \in \mathit{alphabet}\ t \wedge$
$\mathit{depth}\ t\ c = \mathit{height}\ t \wedge \mathit{depth}\ t\ d = \mathit{height}\ t \wedge c \neq d \longrightarrow$
$\mathit{cost}\ (\mathit{swapFourSyms}\ t\ a\ b\ c\ d) \leq \mathit{cost}\ t$

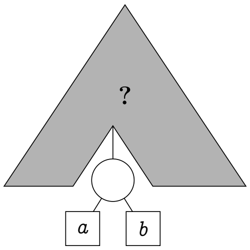
Proof by case analysis on $a = c$, $a = d$, $b = c$ and $b = d$. The cases are easy to prove by expanding the definition of *swapFourSyms* and applying Lemma 26.1. □

The tree *splitLeaf* $t\ w_a\ a\ w_b\ b$ is optimum if t is optimum, under a few assumptions, notably that *freq* $t\ a = w_a + w_b$. Graphically:



Lemma 26.5. $\mathit{consistent}\ t \wedge \mathit{optimum}\ t \wedge$
$a \in \mathit{alphabet}\ t \wedge b \notin \mathit{alphabet}\ t \wedge \mathit{freq}\ t\ a = w_a + w_b \wedge$
$(\forall c{\in}\mathit{alphabet}\ t.\ w_a \leq \mathit{freq}\ t\ c \wedge w_b \leq \mathit{freq}\ t\ c) \longrightarrow$
$\mathit{optimum}\ (\mathit{splitLeaf}\ t\ w_a\ a\ w_b\ b)$

Proof. We assume that t's cost is less than or equal to that of any other comparable tree v and show that *splitLeaf* $t\ w_a\ a\ w_b\ b$ has a cost less than or equal to that of any other comparable tree u. For the nontrivial case where *height* $t > 0$, it is easy to prove that there must be two symbols c and d occurring in sibling nodes at the very bottom of u. From u, we construct the tree *swapFourSyms* $u\ a\ b\ c\ d$ in which the minima a and b are siblings:



The question mark reminds us that we hardly know anything about u's structure. Merging a and b gives a tree comparable with t, which we can use to instantiate v:

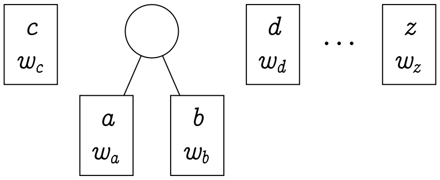
$$\begin{aligned}
&\textit{cost}\ (\textit{splitLeaf}\ t\ a\ w_a\ b\ w_b) = \textit{cost}\ t + w_a + w_b && \text{by Lemma 26.3}\\
&\le \textit{cost}\ (\textit{mergeSibling}\ (\textit{swapFourSyms}\ u\ a\ b\ c\ d)\ a) + w_a + w_b \\
&&& \text{by optimality assumption}\\
&= \textit{cost}\ (\textit{swapFourSyms}\ u\ a\ b\ c\ d) && \text{by Lemma 26.2}\\
&\le \textit{cost}\ u && \text{by Lemma 26.4} \quad \square
\end{aligned}$$

Once it has combined two lowest-weight trees using *uniteTrees*, Huffman's algorithm does not visit these trees ever again. This suggests that splitting a leaf node before applying the algorithm should give the same result as applying the algorithm first and splitting the leaf node afterward.

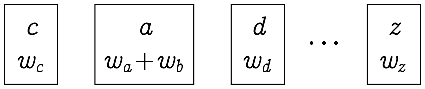
Lemma 26.6.
$\textit{consistent}_F\ ts \land ts \ne [] \land a \in \textit{alphabet}_F\ ts \land \textit{freq}_F\ ts\ a = w_a + w_b \longrightarrow$
$\textit{splitLeaf}\ (\textit{huffman}\ ts)\ w_a\ a\ w_b\ b = \textit{huffman}\ (\textit{splitLeaf}_F\ ts\ w_a\ a\ w_b\ b)$

The proof is by straightforward induction on the length of the forest ts.

As a consequence of this commutativity lemma, applying Huffman's algorithm on a forest of the form



gives the same result as applying the algorithm on the "flat" forest

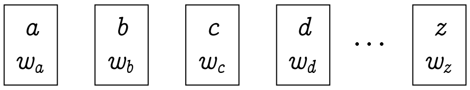


followed by splitting the leaf node a into two nodes a and b with frequencies w_a, w_b. The lemma provides a way to flatten the forest at each step of the algorithm.

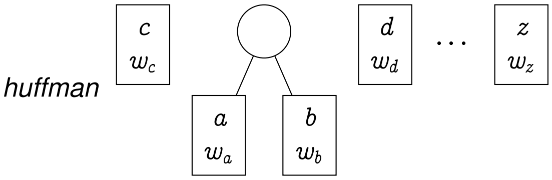
This leads us to our main result.

Theorem 26.7.
$\mathit{consistent}_F\ ts \wedge \mathit{height}_F\ ts = 0 \wedge \mathit{sortedByWeight}\ ts \wedge ts \neq [] \longrightarrow$
$\mathit{optimum}\ (\mathit{huffman}\ ts)$

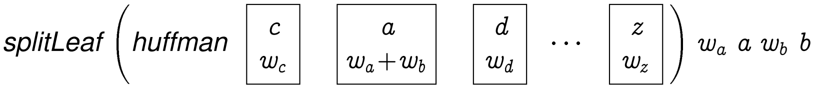
Proof by induction on the length of ts. The assumptions ensure that ts is of the form



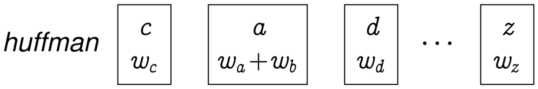
with $w_a \leq w_b \leq w_c \leq w_d \leq \cdots \leq w_z$. If ts consists of a single node, the node has cost 0 and is therefore optimum. If ts has length 2 or more, the first step of the algorithm leaves us with a term such as



In the diagram, we put the newly created tree at position 2 in the forest; in general, it could be anywhere. By Lemma 26.6, the above tree equals



To prove that this tree is optimum, it suffices by Lemma 26.5 to show that



is optimum, which follows from the induction hypothesis. □

In summary, we have established that the *huffman* program, which constitutes a functional implementation of Huffman's algorithm, constructs a binary tree that represents an optimum binary code for the specified alphabet and frequencies.

Chapter Notes

The sorted list of trees constitutes a simple priority queue (Part III). The time complexity of Huffman's algorithm is quadratic in the size n of this queue. By using a binary search to implement *insortTree*, we can obtain an $O(n \lg n)$ imperative implementation. An $O(n)$ implementation is possible by maintaining two queues, one containing the unprocessed leaf nodes and the other containing the combined trees [Knuth 1997].

Huffman's algorithm was invented by Huffman [1952]. The proof above was inspired by Knuth's informal argument [Knuth 1997]. This chapter's text is based on a published article [Blanchette 2009], with the publisher's permission. An alternative formal proof, developed using Coq, is due to Théry [2004].

Knuth [1982] presented an alternative, more abstract view of Huffman's algorithm as a "Huffman algebra." Could his approach help simplify our proof? The most tedious steps above concerned splitting nodes, merging siblings, and swapping symbols. These steps would still be necessary as the algebraic approach seems restricted to abstracting over the arithmetic reasoning, which is not very difficult in the first place. On the other hand, with Knuth's approach, perhaps the proof would gain in elegance.

27 Alpha-Beta Pruning

Tobias Nipkow

This chapter is about searching for the best possible move in a game tree. Alpha-beta pruning is a technique for decreasing the number of nodes that need to be examined by discarding whole subtrees during the search. There are many variations on this theme and we progress from the simple to the more sophisticated.

27.1 Game Trees and Their Evaluation

A **game tree** represents a two-player game, such as tic-tac-toe or chess. Each node in the tree represents a possible **position** in the game. Each **move** is represented by an edge from one position to a child node, the successor position. There may be any finite number of successor positions and thus children. An example game tree is shown in Figure 27.1. In a two-player game, the players take turns. Thus each level in the tree is associated with one of the two players, the one who is about to move, and this

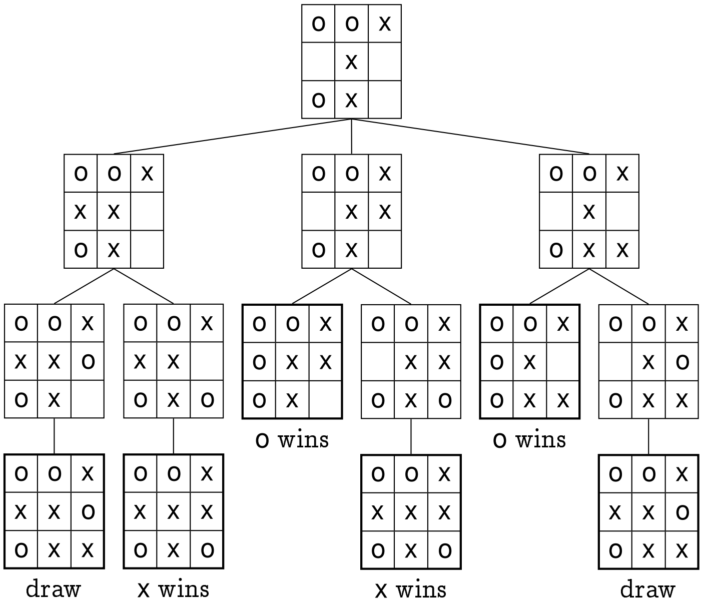


Figure 27.1 Tic-tac-toe game tree

alternates from level to level. Leaf nodes in a game tree are terminal positions. The rules of the game must determine the outcome at a leaf, i.e. who has won or if it is a draw. More generally, what the value of that leaf is, because the game might involve, for example, money that one player loses and the other wins.

We model game trees by the following datatype:

```
datatype 'a tree = Lf 'a | Nd ('a tree list)
```

The interpretation: $'a$ is the type of values, *Lf* v is a leaf of value v and *Nd* ts is a node with a list of successor nodes ts. In an induction on *tree*s, the induction step needs to prove P (*Nd* ts) under the IH that P is true for all t in ts: $\forall t \in set\ ts.\ P\ t$.

Usually the type of values is fixed to be some numeric type extended with ∞ and $-\infty$, e.g. the extended real numbers (type *ereal* in Isabelle). Instead, we will only assume that $'a$ is a linear order with least and greatest elements $\bot$ and $\top$:

$$\bot \le a \qquad a \le \top$$

This is a **bounded linear order**. Until further notice we assume that $'a$ is a bounded linear order. For concreteness, the reader is welcome to think in terms of some extended numeric type.

Type *tree* is an abstraction of an actual game tree (as in Figure 27.1) because the positions are not part of the tree. This is justified because we will only be interested in the value of a game tree, not the positions within it. Given a game tree, we want to find the best move for the start player, i.e. which of its successor nodes it should move to. Essentially equivalent is the question of the **value** of the game tree. This is the highest value of all leaves that the start player can reach, no matter what the opponent does, who will try to thwart those efforts as best as it can. Formally, there is a maximizing and a minimizing player. Thus the value of a game tree depends on who is about to move. Function *maxmin* maximizes and *minmax* minimizes:

```
maxmin :: 'a tree ⇒ 'a
maxmin (Lf x) = x
maxmin (Nd ts) = maxs (map minmax ts)

minmax :: 'a tree ⇒ 'a
minmax (Lf x) = x
minmax (Nd ts) = mins (map maxmin ts)
```

```
maxs :: 'a list ⇒ 'a
maxs [] = ⊥
maxs (x # xs) = max x (maxs xs)

mins :: 'a list ⇒ 'a
mins [] = ⊤
mins (x # xs) = min x (mins xs)
```

The two evaluation functions *maxmin* and *minmax* should be considered the (executable) specification of what this chapter is about, namely more efficient evaluation functions that do not always examine the whole tree.

Figure 27.2 shows a game tree where each node is labeled with its value. The final level are the leaves. The squares are maximizing nodes, the circles are minimizing nodes. The value 3 at the root shows that the maximizer can reach a leaf of value at least 3, no matter which moves the minimizer chooses.

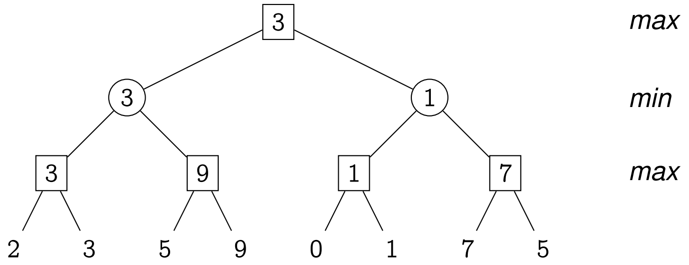


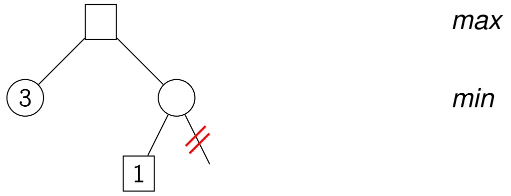
Figure 27.2 Game tree evaluation with *maxmin*

It is usually impossible to build a complete game tree because it is too large. Therefore the tree is typically only built up to some (possibly variable) depth. For simplicity we do not model this building process but start from the generated game tree where the leaves are not necessarily terminal positions (whose value would be determined by the rules of the game) but arbitrary ones where the tree building has stopped (e.g. due to some depth limit) and the value is given by some heuristic evaluation function. However, by starting with a game tree we abstract from all of these issues.

27.2 Alpha-Beta Pruning

27.2.1 Intuition

Consider this partially evaluated game tree:



After we have determined the values 3 and 1, there is no need to evaluate further children of node ○ because the maximizer would never move from the root to this node because then the minimizer could achieve 1, whereas the maximizer can already ensure 3 by moving to the first successor of the root.

In general: If we already have a bound a on the value of node n_1 (belonging to player 1) and are exploring a successor node n_2 of n_1 (belonging to player 2), we can stop exploring the successors of n_2 once we have found a successor that permits player 2 to achieve a better (from its perspective) value than a: player 1 would never move to n_2 because it can achieve the better (for itself) value a elsewhere.

This situation is found twice in Figure 27.3 (the tree with the by now familiar leaf sequence, but evaluated with alpha-beta pruning; ignore the a,b labels for now), once for the maximizer and once for the minimizer.

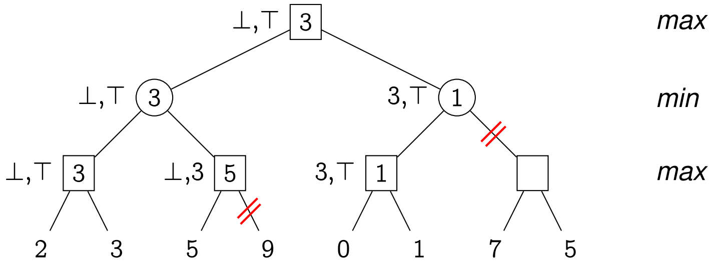
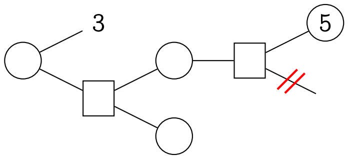


Figure 27.3 Alpha-beta pruning

In contrast to the examples seen so far, pruning may happen at arbitrarily deep levels below the node where the bound (here: 3) comes from:



27.2.2 Implementation

Alpha-beta pruning is parameterized by two bounds a and b (or α and β) where a is the maximum value that the maximizer is already assured of and b is the minimum value that the minimizer is already assured of (by the search so far, assuming optimal play by both players). The maximizer searches its successor positions and increases a accordingly. Once $a \geq b$, the search at this level can stop: if $a > b$, the minimizer would never allow the maximizer to reach the parent node because the minimizer can already enforce b elsewhere; if $a = b$, the minimizer will only allow the maximizer to reach the parent node if the remaining successor positions do not yield a value $> a$. In summary, the open interval from a to b is the window in which alpha-beta pruning searches for nodes that increase a until the interval becomes empty. Dually for the minimizer. This is the actual code:

```
ab_max :: 'a ⇒ 'a ⇒ 'a tree ⇒ 'a
ab_max _ _ (Lf x) = x
ab_max a b (Nd ts) = ab_maxs a b ts

ab_maxs :: 'a ⇒ 'a ⇒ 'a tree list ⇒ 'a
ab_maxs a _ [] = a
ab_maxs a b (t # ts)
= (let a′ = max a (ab_min a b t) in if b ≤ a′ then a′ else ab_maxs a′ b ts)

ab_min :: 'a ⇒ 'a ⇒ 'a tree ⇒ 'a
ab_min _ _ (Lf x) = x
ab_min a b (Nd ts) = ab_mins a b ts

ab_mins :: 'a ⇒ 'a ⇒ 'a tree list ⇒ 'a
ab_mins _ b [] = b
ab_mins a b (t # ts)
= (let b′ = min b (ab_max a b t) in if b′ ≤ a then b′ else ab_mins a b′ ts)
```

Figure 27.3 shows the behaviour of alpha-beta pruning on our example game tree. Each node is annotated with the a,b values with which it is searched and with the final value returned at the end of the search.

There are more compact ways to formulate these functions (Exercise 27.2) but the explicitness of the above code leads to more elementary proofs where the *min* cases are completely dual to the *max* cases. If we only consider one of the two cases in a definition, a lemma or a proof, the other one is completely dual. An example is this simple inductive property of *ab_maxs*

$$a \leq \textit{ab\_maxs}\ a\ b\ ts \tag{27.1}$$

where we leave the dual property of *ab_mins* unstated.

Many properties of alpha-beta pruning require $a < b$, property (27.1) being an exception.

27.2.3 Correctness and Proof

This is the top-level correctness property we want in the end:

$$\textit{ab\_max}\ \bot\ \top\ t = \textit{maxmin}\ t \tag{27.2}$$

Of course, a proof will require a generalization from $\bot$ and $\top$ to arbitrary a and b. Unsurprisingly, $\textit{ab\_max}\ a\ b\ t = \textit{maxmin}\ t$ does not hold in general. Thus we first need to find a suitable generalization of (27.2).

The following relations between *ab_max* and *maxmin* state that *ab_max* coincides with *maxmin* for values inside the (a,b) interval and that *ab_max* bounds *maxmin* outside that interval:

$$\textit{ab\_max}\ a\ b\ t \leq a \quad\longrightarrow\quad \textit{maxmin}\ t \leq \textit{ab\_max}\ a\ b\ t \tag{27.3}$$

$$a < \textit{ab\_max}\ a\ b\ t < b \quad\longrightarrow\quad \textit{ab\_max}\ a\ b\ t = \textit{maxmin}\ t \tag{27.4}$$

$$\textit{ab\_max}\ a\ b\ t \geq b \quad\longrightarrow\quad \textit{maxmin}\ t \geq \textit{ab\_max}\ a\ b\ t \tag{27.5}$$

These properties do not specify *ab_max* uniquely but they are strong enough to imply (as we see below) the key correctness property (27.2).

To facilitate the further discussion, we define the following abbreviation:

$$\begin{aligned}
&ab \leq v\ (\mathrm{mod}\ a,b) \equiv\\
&((ab \leq a \longrightarrow v \leq ab)\ \wedge\\
&(a < ab \wedge ab < b \longrightarrow ab = v)\ \wedge\\
&(b \leq ab \longrightarrow ab \leq v))
\end{aligned} \tag{27.6}$$

The conjunction of (27.3)–(27.5) is $\textit{ab\_max}\ a\ b\ t \leq \textit{maxmin}\ t\ (\mathrm{mod}\ a,b)$. The notation $ab \leq v\ (\mathrm{mod}\ a,b)$ symbolizes that ab is closer to the interval (a,b) than v (or they are equal).

Although "$\leq$ mod" is a relation, it can also be read as a function that tells us in which of the three intervals (not lists!) $[\bot,ab]$, $[ab,ab]$ or $[ab,\top]$ v is located, depending on where ab lies w.r.t. a and b.

Correctness can now be shown simultaneously for all four functions:

Theorem 27.1.

$$
\begin{aligned}
&a < b \longrightarrow \mathit{ab\_max}\ a\ b\ t \le \mathit{maxmin}\ t \ (\mathrm{mod}\ a{,}b) && (27.7)\\
&a < b \longrightarrow \mathit{ab\_maxs}\ a\ b\ ts \le \mathit{maxmin}\ (\mathit{Nd}\ ts)\ (\mathrm{mod}\ a{,}b)\\
&a < b \longrightarrow \mathit{ab\_min}\ a\ b\ t \le \mathit{minmax}\ t \ (\mathrm{mod}\ a{,}b)\\
&a < b \longrightarrow \mathit{ab\_mins}\ a\ b\ ts \le \mathit{minmax}\ (\mathit{Nd}\ ts)\ (\mathrm{mod}\ a{,}b)
\end{aligned}
$$

Proof by simultaneous induction on the computation of $\mathit{ab\_max}$ and friends. The only two nontrivial cases are the ones stemming from the recursion equations for $\mathit{ab\_maxs}$ and $\mathit{ab\_mins}$. We concentrate on $\mathit{ab\_maxs}$. For succinctness we introduce the following abbreviations:

$$
\begin{array}{lll}
abt \equiv \mathit{ab\_min}\ a\ b\ t & abts \equiv \mathit{ab\_maxs}\ a'\ b\ ts & a' \equiv \mathit{max}\ a\ abt\\
vt \equiv \mathit{minmax}\ t & vts \equiv \mathit{maxmin}\ (\mathit{Nd}\ ts) &
\end{array}
$$

The two IHs are

$$
\begin{aligned}
&abt \le vt\ (\mathrm{mod}\ a{,}b) && \text{(IH1)}\\
&a' < b \longrightarrow abts \le vts\ (\mathrm{mod}\ a'{,}b) && \text{(IH2)}
\end{aligned}
$$

and we need to prove $abtts \le vtts\ (\mathrm{mod}\ a{,}b)$ where

$$
\begin{aligned}
&abtts \equiv \mathit{ab\_maxs}\ a\ b\ (t\ \#\ ts)\\
&vtts \equiv \mathit{maxmin}\ (\mathit{Nd}\ (t\ \#\ ts)) = \mathit{max}\ vt\ vts
\end{aligned}
$$

We focus on the most complex part of $abtts \le vtts\ (\mathrm{mod}\ a{,}b)$, conjunct 2. That is, we assume $a < abtts < b$ and prove $abtts = vtts$ by case analysis. The case $b \le a'$ is impossible because it would imply $a' = abtts$, which, combined with the assumption $abtts < b$, would imply $b < b$. Hence we can assume $a' < b$ and thus $abtts = abts$ and $a < abts < b$. Hence we now need to prove

$$abts = \mathit{max}\ vt\ vts$$

For the following detailed arguments we display and name the relevant conjuncts of IH1 and IH2 (where the premise $a' < b$ is now assumed):

$$
\begin{aligned}
&abt \le a && \longrightarrow vt \le abt && \text{(IH11)}\\
&a < abt < b && \longrightarrow abt = vt && \text{(IH12)}\\
&abts \le a' && \longrightarrow vts \le abts && \text{(IH21)}\\
&a' < abts < b && \longrightarrow abts = vts && \text{(IH22)}
\end{aligned}
$$

The proof continues with a case analysis. First assume $abt \le a$. Hence $a' = a$ and thus IH22 and $a < abts < b$ yield $abts = vts$. Moreover, $vt \le vts$ follows from IH11, $abt \le a$, $a < abts$ and $abts = vts$. Together this proves $abts = \mathit{max}\ vt\ vts$.

Now assume $a < abt$. This implies $a' = abt$, $abt = vt$ (using IH12) and $abt < b$ (using $a' < b$). From (27.1) we obtain $a' \le abts$ and perform another case analysis.

First assume $a' < abts$. Because $abts < b$, IH22 yields $abts = vts$. Assumption $a' < abts$ implies $abt < abts$ and thus $vt < vts$ which proves $abts = \mathsf{max}\ vt\ vts$. Now assume $a' = abts$. IH21 implies $vts \leq abts$. Moreover, $abts = a' = abt = vt$. Together this implies $abts = \mathsf{max}\ vt\ vts$. □

The top-level correctness property $\mathsf{ab\_max}\ \bot\ \top\ t = \mathsf{maxmin}\ t$ (27.2) is a consequence of (27.7) where $a = \bot$ and $b = \top$. Let us first deal with the standard case that $\bot < \top$. Then (27.7) yields $\mathsf{ab\_max}\ a\ b\ t \leq \mathsf{maxmin}\ t\ (\mathrm{mod}\ a,b)$. The claim $\mathsf{ab\_max}\ \bot\ \top\ t = \mathsf{maxmin}\ t$ follows from this general property of "$\leq$ mod"

$$y \leq x\ (\mathrm{mod}\ \bot,\top) \longrightarrow x = y$$

which is easy to prove: If $\bot < y < \top$, the definition yields the result directly. If $y \leq \bot$ then the definition implies $x \leq y$ and uniqueness of $\bot$ yields $x = y$ $(= \bot)$. The case $y \geq \top$ is dual.

Now consider the corner case which does not arise for numeric types, namely $\neg \bot < \top$ In that case, everything collapses (exercise!)

$$\neg \bot < \top \longrightarrow x = y$$

and (27.2) trivially holds.

27.2.4 Fail-Soft

Function $\mathsf{ab\_maxs}$ is less precise than it could be: $\mathsf{ab\_maxs}\ a\ b\ ts = a$ even if $\mathsf{ab\_min}\ a\ b\ t < a$ for all $t \in \mathsf{set}\ ts$. But in this case $\mathsf{maxmin}\ (\mathsf{Nd}\ ts) < a$ and $\mathsf{ab\_maxs}$ could have produced a better bound for $\mathsf{maxmin}\ (\mathsf{Nd}\ ts)$ if it did not return a but $\bot$ at the end of the list. These are the improved $\mathsf{ab\_max}$ functions:

```
ab_max' :: 'a ⇒ 'a ⇒ 'a tree ⇒ 'a
ab_max' _ _ (Lf x) = x
ab_max' a b (Nd ts) = ab_maxs' a b ⊥ ts

ab_maxs' :: 'a ⇒ 'a ⇒ 'a ⇒ 'a tree list ⇒ 'a
ab_maxs' _ _ m [] = m
ab_maxs' a b m (t # ts)
= (let m' = max m (ab_min' (max m a) b t)
   in if b ≤ m' then m' else ab_maxs' a b m' ts)
```

In the literature, $\mathsf{ab\_maxs}$ is called the **fail-hard** variant (because it brutally cuts off at a) and $\mathsf{ab\_maxs}'$ the **fail-soft** variant (because it "fails" more gracefully).

For a start we have that $\mathit{ab\_max}'$ bounds maxmin (and is thus correct w.r.t. maxmin):

Theorem 27.2. $a < b \longrightarrow \mathit{ab\_max}' \; a \; b \; t \le \mathit{maxmin} \; t$ (mod a,b)
$\mathit{max} \; m \; a < b \longrightarrow \mathit{ab\_maxs}' \; a \; b \; m \; ts \le \mathit{maxmin} \; (\mathit{Nd} \; ts)$ (mod $\mathit{max} \; m \; a$,b)

This is similar to the correctness theorem for $\mathit{ab\_max}$ but slightly more involved because of the additional parameter of $\mathit{ab\_max}'$. The proof is also similar, including the need for the lemmas $m \le \mathit{ab\_maxs}' \; a \; b \; m \; ts$ and $\mathit{ab\_mins}' \; a \; b \; m \; ts \le m$.

Moreover, $\mathit{ab\_max}$ bounds $\mathit{ab\_max}'$:

Theorem 27.3. $a < b \longrightarrow \mathit{ab\_max} \; a \; b \; t \le \mathit{ab\_max}' \; a \; b \; t$ (mod a,b)
$\mathit{max} \; m \; a < b \longrightarrow \mathit{ab\_maxs} \; (\mathit{max} \; m \; a) \; b \; ts \le \mathit{ab\_maxs}' \; a \; b \; m \; ts$ (mod a,b)

The proof is similar to that of the previous theorem but requires no lemmas.

In summary, we now know that $\mathit{ab\_max}'$ bounds maxmin at least as precisely as $\mathit{ab\_max}$ does. In fact, it can be more precise, as the following example shows: $\mathit{ab\_max}' \; 0 \; 1 \; (\mathit{Nd} \; []) = \mathit{maxmin} \; (\mathit{Nd} \; []) = \bot$ but $\mathit{ab\_max} \; 0 \; 1 \; (\mathit{Nd} \; []) = 0 > \bot$.

Both variants search the same part of the trees. To verify this, we define functions that return the part of the trees that $\mathit{ab\_max}$(') and $\mathit{ab\_maxs}$(') traverse.

```
abt_max :: 'a ⇒ 'a ⇒ 'a tree ⇒ 'a tree
abt_max _ _ (Lf x) = Lf x
abt_max a b (Nd ts) = Nd (abt_maxs a b ts)

abt_maxs :: 'a ⇒ 'a ⇒ 'a tree list ⇒ 'a tree list
abt_maxs _ _ [] = []
abt_maxs a b (t # ts)
= (let u = abt_min a b t; a' = max a (ab_min a b t)
   in u # (if b ≤ a' then [] else abt_maxs a' b ts))

abt_max' :: 'a ⇒ 'a ⇒ 'a tree ⇒ 'a tree
abt_max' _ _ (Lf x) = Lf x
abt_max' a b (Nd ts) = Nd (abt_maxs' a b ⊥ ts)

abt_maxs' :: 'a ⇒ 'a ⇒ 'a ⇒ 'a tree list ⇒ 'a tree list
abt_maxs' _ _ _ [] = []
abt_maxs' a b m (t # ts)
= (let u = abt_min' (max m a) b t; m' = max m (ab_min' (max m a) b t)
   in u # (if b ≤ m' then [] else abt_maxs' a b m' ts))
```

Indeed, they search the same part of the trees:

Theorem 27.4. $a < b \longrightarrow \mathsf{abt\_max}'\ a\ b\ t = \mathsf{abt\_max}\ a\ b\ t$
$\mathsf{max}\ m\ a < b \longrightarrow \mathsf{abt\_maxs}'\ a\ b\ m\ ts = \mathsf{abt\_maxs}\ (\mathsf{max}\ m\ a)\ b\ ts$

The proof is the usual simultaneous induction and relies on Theorem 27.3.

The following section answers the question how the improved precision of the soft variant can be exploited to optimize the search further.

27.2.5 From Trees to Graphs

Game trees are in fact graphs, because different paths may lead to the same position. Moreover, positions have symmetries, and different positions may be equivalent, for example by rotating or reflecting the board. For efficiency reasons it is vital to factor in these symmetries when searching the graph. This is usually taken care of by a so-called **transposition table**, which is a cache for storing evaluations of previously seen positions (modulo symmetries). However, evaluations of the same position from different parts of the graph typically come with different a,b windows. Nevertheless, the result of a previous evaluation can help to narrow the a,b window in later evaluations of the same position. In the following little lemma, we assume that abf :: $'a \Rightarrow {'a} \Rightarrow {'a}\ tree \Rightarrow {'a}$ is some function (e.g. $\mathsf{ab\_max}'$) that bounds maxmin:

$$\forall a\ b.\ abf\ a\ b\ t \le \mathsf{maxmin}\ t\ (\mathrm{mod}\ a{,}b) \qquad (*)$$

If in a previous call $b \le abf\ a\ b\ t$, then $(*)$ implies $abf\ a\ b\ t \le \mathsf{maxmin}\ t$. Thus $abf\ a\ b\ t$ can be used as a lower bound for future abf calls. That is, in a call $abf\ a'\ b'\ t$ we can replace a' by $\mathsf{max}\ a'\ (abf\ a\ b\ t)$, provided this does not push us above b' (in which case there is no need to call abf again):

$$\begin{array}{l} b \le abf\ a\ b\ t \wedge \mathsf{max}\ a'\ (abf\ a\ b\ t) < b' \longrightarrow \\ abf\ (\mathsf{max}\ a'\ (abf\ a\ b\ t))\ b'\ t \le \mathsf{maxmin}\ t\ (\mathrm{mod}\ a'{,}b') \end{array}$$

Similarly, if $abf\ a\ b\ t \le a$, then $abf\ a\ b\ t$ can be used as an upper bound for future abf calls, i.e. we can replace b' by $\mathsf{min}\ b'\ (abf\ a\ b\ t)$. Hence $\mathsf{ab\_max}'$ has the edge over $\mathsf{ab\_max}$ in this scenario: it can lead to smaller search windows.

Of course, if $a < abf\ a\ b\ t < b$, then $abf\ a\ b\ t = \mathsf{maxmin}\ t$ and we can return the exact value right away.

The advantage of narrowing the a,b window is that the search space decreases. The intuitive reason is clear: as b decreases, a will reach b more quickly (and conversely). More precisely, the search space with a smaller window is a prefix of that with the larger window in the following sense:

```
prefix :: 'a tree ⇒ 'a tree ⇒ bool
prefix (Lf x) (Lf y) = (x = y)
prefix (Nd ts) (Nd us) = prefixs ts us
prefix _ _ = False

prefixs :: 'a tree list ⇒ 'a tree list ⇒ bool
prefixs [] _ = True
prefixs (t # ts) (u # us) = (prefix t u ∧ prefixs ts us)
prefixs (_ # _) [] = False
```

Now we can employ the *abt_* functions (Section 27.2.4) to obtain the searched space:

Theorem 27.5.
$a < b \wedge a' \le a \wedge b \le b' \longrightarrow \mathit{prefix}\ (\mathit{abt\_max}'\ a\ b\ t)\ (\mathit{abt\_max}'\ a'\ b'\ t)$
$\mathit{max}\ m\ a < b \wedge a' \le a \wedge b \le b' \wedge m' \le m \longrightarrow$
$\mathit{prefixs}\ (\mathit{abt\_maxs}'\ a\ b\ m\ ts)\ (\mathit{abt\_maxs}'\ a'\ b'\ m'\ ts)$

The proof is by the usual computation induction but also requires a lemma. It expresses that when we narrow the search window, the result becomes less precise:

Lemma 27.6.
$a < b \wedge a' \le a \wedge b \le b' \longrightarrow \mathit{ab\_max}'\ a\ b\ t \le \mathit{ab\_max}'\ a'\ b'\ t\ (\mathrm{mod}\ a,b)$
$\mathit{max}\ m\ a < b \wedge a' \le a \wedge b \le b' \wedge m' \le m \longrightarrow$
$\mathit{ab\_maxs}'\ a\ b\ m\ ts \le \mathit{ab\_maxs}'\ a'\ b'\ m'\ ts\ (\mathrm{mod}\ \mathit{max}\ m\ a,b)$

This lemma can be proved directly, i.e. without requiring further lemmas.

27.2.6 Exercises

Exercise 27.1. We can get away without $\bot$ and $\top$ if we require that the list of successor positions, i.e. the arguments of *Nd*, are nonempty. Formalize this requirement as a predicate $\mathit{invar} :: {'a}\ \mathit{tree} \Rightarrow \mathit{bool}$, define new versions of *maxs*, *mins*, *maxmin* and *minmax* (without using $\bot$ and $\top$!) and prove $\mathit{invar}\ t \longrightarrow \mathit{maxmin1}\ t = \mathit{maxmin}\ t$ (where the new versions are distinguished by an appended 1).

Exercise 27.2. The functions *ab_max*/*ab_min* and the functions *ab_maxs*/*ab_mins* are completely dual to each other. Similarly for *maxmin*/*minmax*. Eliminate this duplication by defining uniform versions of these functions that are suitably parameterized (e.g. by the ordering or by a boolean flag) and can play both the *min* and the *max* part. Prove that the uniform functions (with the right arguments) are equal to the corresponding old functions.

Exercise 27.3. Prove that "$\leq$ mod" (27.6) can be expressed as follows if $a < b$:

$$ab \leq v \ (\text{mod } a,b) \longleftrightarrow \mathit{min}\ v\ b \leq ab \wedge ab \leq \mathit{max}\ v\ a$$

Exercise 27.4. Consider this weaker version of "$\leq$ mod":

$$x \cong y \ (\text{mod } a,b) \equiv$$
$$((y \leq a \longrightarrow x \leq a) \wedge (a < y \wedge y < b \longrightarrow y = x) \wedge (b \leq y \longrightarrow b \leq x))$$

Again we have $x \cong y \ (\text{mod } \bot,\top) \longrightarrow x = y$. Prove

$$a < b \longrightarrow \mathit{maxmin}\ t \cong \mathit{ab\_max}\ a\ b\ t \ (\text{mod } a,b)$$

following the proof of Theorem 27.1. Do not simply employ that $y \leq x \ (\text{mod } a,b)$ implies $x \cong y \ (\text{mod } a,b)$.

Exercise 27.5. Consider the operation $\mathit{max}\ a\ (\mathit{min}\ x\ b)$ that squashes x into the closed interval $[a,b]$ (assuming $a \leq b$) by returning a if $x < a$ and b if $x > b$ and leaving x unchanged otherwise. Note that if $a \leq b$, then the order of *max* and *min* is irrelevant: $a \leq b \longrightarrow \mathit{max}\ a\ (\mathit{min}\ x\ b) = \mathit{min}\ b\ (\mathit{max}\ x\ a)$.

Prove that with the help of this operation, $\cong$ (see Exercise 27.4) can be expressed purely equationally if $a < b$:

$$x \cong y \ (\text{mod } a,b) \longleftrightarrow \mathit{max}\ a\ (\mathit{min}\ x\ b) = \mathit{max}\ a\ (\mathit{min}\ y\ b)$$

Because the right-hand side is symmetric in x and y, it follows that $\cong$ is symmetric as well: $a < b \longrightarrow x \cong y \ (\text{mod } a,b) \longleftrightarrow y \cong x \ (\text{mod } a,b)$.

Exercise 27.6. Consider the $\mathit{max}\ a\ (\mathit{min}\ x\ b)$ operation from Exercise 27.5 and modify $\mathit{ab\_max}(s)$ (and analogously $\mathit{ab\_min}(s)$) as follows:

$$\mathit{ab\_max2} :: 'a \Rightarrow 'a \Rightarrow 'a\ \mathit{tree} \Rightarrow 'a$$
$$\mathit{ab\_max2}\ a\ b\ (\mathit{Lf}\ x) = \mathit{max}\ a\ (\mathit{min}\ x\ b)$$
$$\mathit{ab\_max2}\ a\ b\ (\mathit{Nd}\ ts) = \mathit{ab\_maxs2}\ a\ b\ ts$$

$$\mathit{ab\_maxs2} :: 'a \Rightarrow 'a \Rightarrow 'a\ \mathit{tree\ list} \Rightarrow 'a$$
$$\mathit{ab\_maxs2}\ a\ \_\ [] = a$$
$$\mathit{ab\_maxs2}\ a\ b\ (t\ \#\ ts)$$
$$= (\textbf{let}\ a' = \mathit{ab\_min2}\ a\ b\ t\ \textbf{in}\ \textbf{if}\ a' = b\ \textbf{then}\ a'\ \textbf{else}\ \mathit{ab\_maxs2}\ a'\ b\ ts)$$

Both *max* and *min* have moved to the *Lf* cases, thus assuring that the result of all *ab_* functions lies in the closed interval $[a,b]$. Prove the following correctness theorem

$$a \leq b \longrightarrow \mathit{ab\_max2}\ a\ b\ t = \mathit{max}\ a\ (\mathit{min}\ (\mathit{maxmin}\ t)\ b)$$

The corollary $\mathit{ab\_max2}\ \bot\ \top\ t = \mathit{maxmin}\ t$ is immediate.

27.3 Negative Values

In this section we examine a popular approach to exploiting the symmetries between maximizer and minimizer. As a result, we only need two instead of four functions, both for game tree evaluation and alpha-beta pruning. It can be seen as another variation of the approaches sketched in Exercise 27.2. This time we exploit the symmetries between positive and negative values. A value v for one player can be viewed as a value $-v$ for the other player: one player's gain is the other player's loss. This seems to work only for numeric value types, but it turns out that the following properties are sufficient to make it work more generally:

$$- (- x) = x$$

$$- \mathit{min}\ x\ y = \mathit{max}\ (- x)\ (- y) \tag{27.8}$$

We call a bounded linear order satisfying the above two properties a **De Morgan order** because of the second, De-Morgan-like property. For the rest of this section, we assume that $'a$ is a De Morgan order. For concreteness you may think of the extended reals. Of course De Morgan orders satisfy many other properties that follow easily, in particular the dual De Morgan property

$$- \mathit{max}\ x\ y = \mathit{min}\ (- x)\ (- y)$$

We will not list them here because they are all familiar from extended numeric types.

27.3.1 Game Tree Evaluation

With the help of negation we can unify the evaluation functions *maxmin* and *minmax* into a single function *negmax*:

```
negmax :: 'a tree ⇒ 'a
negmax (Lf x) = x
negmax (Nd ts) = maxs (map (λt. − negmax t) ts)
```

Figure 27.4 shows the evaluation of the same tree as in Figure 27.2 but with *negmax*. We have to negate the leaves because they belong to the minimizer but the root (which we evaluate) belongs to the maximizer.

Function *negate* $b\ t$ performs the negation of the minimizer leaves of t, where $b =$ *True* iff the root of t is a minimizer level:

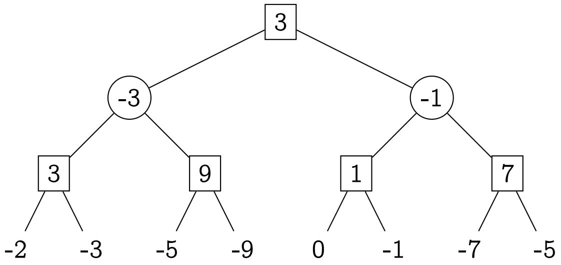


Figure 27.4 Game tree evaluation with *negmax*

$$\mathit{negate} :: \mathit{bool} \Rightarrow {'a}\ \mathit{tree} \Rightarrow {'a}\ \mathit{tree}$$
$$\mathit{negate}\ b\ (\mathit{Lf}\ x) = \mathit{Lf}\ (\textbf{if}\ b\ \textbf{then}\ -\ x\ \textbf{else}\ x)$$
$$\mathit{negate}\ b\ (\mathit{Nd}\ ts) = \mathit{Nd}\ (\mathit{map}\ (\mathit{negate}\ (\neg\ b))\ ts)$$

Now we can express that *negmax* correctly mimics the behaviour of *maxmin* and *minmax*:

$$\mathit{maxmin}\ t = \mathit{negmax}\ (\mathit{negate}\ \mathit{False}\ t) \tag{27.9}$$

$$\mathit{minmax}\ t = -\ \mathit{negmax}\ (\mathit{negate}\ \mathit{True}\ t) \tag{27.10}$$

The proof is by simultaneous induction on the computations of *maxmin* and *minmax*. We focus on the induction step. By IH the equation holds for all $t \in \mathit{set}\ ts$. The IH will be combined with the following general congruence property for *map*:

$$(\forall x {\in} \mathit{set}\ xs.\ f\ x = g\ x) \longrightarrow \mathit{map}\ f\ xs = \mathit{map}\ g\ xs \tag{27.11}$$

The proof of (27.9) follows:

$$\begin{aligned}
&\mathit{maxmin}\ (\mathit{Nd}\ ts) = \mathit{maxs}\ (\mathit{map}\ \mathit{minmax}\ ts)\\
&= \mathit{maxs}\ (\mathit{map}\ (\lambda t.\ -\ \mathit{negmax}\ (\mathit{negate}\ \mathit{True}\ t))\ ts) \qquad \text{by (27.11) and IH}\\
&= \mathit{maxs}\ (\mathit{map}\ ((\lambda t.\ -\ \mathit{negmax}\ t) \circ \mathit{negate}\ \mathit{True})\ ts)\\
&= \mathit{maxs}\ (\mathit{map}\ (\lambda t.\ -\ \mathit{negmax}\ t)\ (\mathit{map}\ (\mathit{negate}\ \mathit{True})\ ts))\\
&\qquad\qquad \text{by } \mathit{map}\ f\ (\mathit{map}\ g\ xs) = \mathit{map}\ (f \circ g)\ xs\\
&= \mathit{negmax}\ (\mathit{Nd}\ (\mathit{map}\ (\mathit{negate}\ \mathit{True})\ ts))\\
&= \mathit{negmax}\ (\mathit{negate}\ \mathit{False}\ (\mathit{Nd}\ ts))
\end{aligned}$$

The proof of (27.10) is almost dual but also uses a generalization of (27.8) to lists, which follows easily by induction:

$$-\ \mathit{mins}\ (\mathit{map}\ f\ xs) = \mathit{maxs}\ (\mathit{map}\ (\lambda x.\ -\ f\ x)\ xs)$$

27.3.2 Alpha-Beta Pruning

Alpha-beta pruning for De Morgan orders is easily derived from the *ab_max*/*min* functions using negation and swapping a and b when switching between players:

```
ab_negmax :: 'a ⇒ 'a ⇒ 'a tree ⇒ 'a
ab_negmax _ _ (Lf x) = x
ab_negmax a b (Nd ts) = ab_negmaxs a b ts

ab_negmaxs :: 'a ⇒ 'a ⇒ 'a tree list ⇒ 'a
ab_negmaxs a _ [] = a
ab_negmaxs a b (t # ts)
= (let a' = max a (- ab_negmax (- b) (- a) t)
   in if b ≤ a' then a' else ab_negmaxs a' b ts)
```

Correctness can be proved easily by simultaneous induction

$a < b \longrightarrow \mathit{ab\_negmax}\ a\ b\ t \le \mathit{negmax}\ t\ (\mathrm{mod}\ a{,}b)$
$a < b \longrightarrow \mathit{ab\_negmaxs}\ a\ b\ ts \le \mathit{negmax}\ (\mathit{Nd}\ ts)\ (\mathrm{mod}\ a{,}b)$

using this simple inductive fact: $a \le \mathit{ab\_negmaxs}\ a\ b\ ts$.

27.3.3 Exercises

Exercise 27.7. It is straightforward to connect *ab_negmax* and *ab_max*

$\mathit{ab\_max}\ a\ b\ t = \mathit{ab\_negmax}\ a\ b\ (\mathit{negate}\ \mathit{False}\ t)$

by simultaneous computation induction involving a further three analogous equations connecting pairs of alpha-beta functions.

Exercise 27.6 carries over to negative values, *mutatis mutandis*.

27.4 Alpha-Beta Pruning for Distributive Lattices

Although alpha-beta pruning is customarily presented for linear orderings, it also works for the more general domain of distributive lattices. This has applications to games with incomplete information such as many card games because distributive lattices can represent sets of possible situations. For games of complete information such as chess, distributive lattices have applications too. They support heuristic evaluations with multiple components (e.g. material, mobility, etc.) without being forced to combine them into a single value or order them linearly because tuples of numbers form a distributive lattice.

27.4.1 Lattices

A **lattice** on some type $'a$ is a partial order $(\leq)$ such that any two elements have a greatest lower and a least upper bound. These two operations are denoted by the following constants and are also called also called **infimum** and **supremum**:

$$(\sqcap) :: {'a} \Rightarrow {'a} \Rightarrow {'a}$$
$$(\sqcup) :: {'a} \Rightarrow {'a} \Rightarrow {'a}$$

They fulfill these properties:

$$x \sqcap y \leq x \qquad x \sqcap y \leq y \qquad x \leq y \wedge x \leq z \longrightarrow x \leq y \sqcap z$$
$$x \leq x \sqcup y \qquad y \leq x \sqcup y \qquad y \leq x \wedge z \leq x \longrightarrow y \sqcup z \leq x$$

That is, $\sqcap$ is the greatest lower and $\sqcup$ the least upper bound. Note that $\sqcap$ has a higher precedence than $\sqcup$: $x \sqcup y \sqcap z$ means $x \sqcup (y \sqcap z)$. Just like $\wedge/\vee$ and $\cap/\cup$.

Any linear order is a lattice where $\sqcap$ = *min* and $\sqcup$ = *max*. An example of a lattice that is not a linear order is the type of sets where $\sqcap = \cap$ and $\sqcup = \cup$.

It turns out that $\sqcap$ and $\sqcup$ have very nice algebraic properties: both are associative and commutative and enjoy these absorption properties:

$$x \sqcap x = x \qquad x \sqcap (x \sqcup y) = x$$
$$x \sqcup x = x \qquad x \sqcup x \sqcap y = x$$

A **distributive lattice** is a lattice where $\sqcap$ and $\sqcup$ distribute over each other:

$$x \sqcup y \sqcap z = (x \sqcup y) \sqcap (x \sqcup z)$$
$$x \sqcap (y \sqcup z) = x \sqcap y \sqcup x \sqcap z$$

Clearly, linear orders and sets form distributive lattices. Moreover, the Cartesian product of distributive lattices is again a distributive lattice.

In the rest of this section we work in a distributive lattice. Often we also assume that the lattice is **bounded**, i.e. has a least and a greatest element $\bot$ and $\top$. Of course bounded lattices satisfy the obvious properties $\bot \sqcap x = \bot$, $\top \sqcap x = x$, $\bot \sqcup x = x$ and $\top \sqcup x = \top$.

In the sequel, we rarely enlarge on parts of a proof that follow by distributive lattice laws alone; we take those for granted. For concreteness the reader may think in terms of sets rather than distributive lattices and will not be misled.

27.4.2 Alpha-Beta Pruning

Both game tree evaluation and alpha-beta pruning are completely analogous to before, except that *min* and *max* are generalized to $\sqcap$ and $\sqcup$. The result is shown in Figure 27.5. We only cover fail-hard here but have also formalized fail-soft.

We will prove *ab_sup* $\bot$ $\top$ t = *supinf* t, but we cannot proceed via the following naive generalization of Theorem 27.1

```
supinf :: 'a tree ⇒ 'a
supinf (Lf x) = x
supinf (Nd ts) = sups (map infsup ts)

infsup :: 'a tree ⇒ 'a
infsup (Lf x) = x
infsup (Nd ts) = infs (map supinf ts)

sups :: 'a list ⇒ 'a
sups [] = ⊥
sups (x # xs) = x ⊔ sups xs

infs :: 'a list ⇒ 'a
infs [] = ⊤
infs (x # xs) = x ⊓ infs xs

ab_sup :: 'a ⇒ 'a ⇒ 'a tree ⇒ 'a
ab_sup _ _ (Lf x) = x
ab_sup a b (Nd ts) = ab_sups a b ts

ab_sups :: 'a ⇒ 'a ⇒ 'a tree list ⇒ 'a
ab_sups a _ [] = a
ab_sups a b (t # ts)
= (let a′ = a ⊔ ab_inf a b t in if b ≤ a′ then a′ else ab_sups a′ b ts)

ab_inf :: 'a ⇒ 'a ⇒ 'a tree ⇒ 'a
ab_inf _ _ (Lf x) = x
ab_inf a b (Nd ts) = ab_infs a b ts

ab_infs :: 'a ⇒ 'a ⇒ 'a tree list ⇒ 'a
ab_infs _ b [] = b
ab_infs a b (t # ts)
= (let b′ = b ⊓ ab_sup a b t in if b′ ≤ a then b′ else ab_infs a b′ ts)
```

Figure 27.5 Game tree evaluation and alpha-beta pruning for lattices

$$a < b \longrightarrow \textit{ab\_sup}\ a\ b\ t \le \textit{supinf}\ t\ (\mathrm{mod}\ a{,}b) \tag{27.12}$$

because it does not hold.

27.4.2.1 Counterexamples

Property (27.12) does not hold in general as the following counterexample for the distributive lattice *bool set* shows. Let $a = \{\mathit{False}\}$, $b = \{\mathit{False}, \mathit{True}\}$ ($a < b$!) and $t = \mathit{Nd}\ [\mathit{Lf}\ \{\mathit{True}\}]$. Then $\mathit{supinf}\ t = \{\mathit{True}\} =: v$ and $\mathit{ab\_sup}\ a\ b\ t = \{\mathit{False}, \mathit{True}\}$ $=: ab$ But although $ab \ge b$, we don't have $v \ge ab$ as (27.12) would require.

More generally, the definition of $ab \le v\ (\mathrm{mod}\ a{,}b)$ implicitly assumes that ab, the result of alpha-beta pruning, satisfies one of the three alternatives $ab \le a$, $a < ab < b$ or $b \le ab$. In a distributive lattice this may no longer be the case. Take $a = \{\}$, $b = \{\mathit{True}\}$ and $t = \mathit{Nd}\ [\mathit{Lf}\ \{\mathit{False}\}]$. Then $\mathit{supinf}\ t = \{\mathit{False}\} =: v$ and $\mathit{ab\_sup}\ a\ b\ t = \{\mathit{True}\} =: ab$. But now all three comparisons $ab \le a$, $a < ab \wedge ab < b$ and $b \le ab$ are false. Thus we cannot draw any conclusion about v from ab.

In summary, for distributive lattices, (27.12) is unsuitable for relating the result of alpha-beta pruning to the true tree value.

27.4.3 Correctness and Proof

We will phrase correctness by means of the operation $a \sqcup x \sqcap b$ that projects ("squashes") x into the closed interval $[a{,}b]$, if $a \le b$:

$$a \le b \longrightarrow a \le a \sqcup x \sqcap b \le b$$

If $a \le x \le b$ then $a \sqcup x \sqcap b = x$. Note also that if $a \le b$, then the order of $\sqcup$ and $\sqcap$ is irrelevant: $a \le b \longrightarrow a \sqcup x \sqcap b = (a \sqcup x) \sqcap b$.

Although $a \sqcup x \sqcap b$ has particularly nice properties if $a \le b$, it can be manipulated algebraically even in the absence of $a \le b$. As an example we have this weak form of the preceding associativity property:

$$a \sqcup x \sqcap b = a \sqcup y \sqcap b \longleftrightarrow (a \sqcup x) \sqcap b = (a \sqcup y) \sqcap b$$

In analogy with $\cong$ (see Exercise 27.5) we define $x \simeq y$ to mean that x and y are the same modulo "squashing":

$$x \simeq y\ (\mathrm{mod}\ a{,}b) \equiv a \sqcup x \sqcap b = a \sqcup y \sqcap b$$

It turns out that the result of alpha-beta pruning is $\simeq$ to the real value. This can be shown simultaneously for all four functions:

Theorem 27.7.
$\mathsf{ab\_sup}\ a\ b\ t \simeq \mathsf{supinf}\ t\ (\mathrm{mod}\ a,b)$
$\mathsf{ab\_sups}\ a\ b\ ts \simeq \mathsf{supinf}\ (\mathsf{Nd}\ ts)\ (\mathrm{mod}\ a,b)$
$\mathsf{ab\_inf}\ a\ b\ t \simeq \mathsf{infsup}\ t\ (\mathrm{mod}\ a,b)$
$\mathsf{ab\_infs}\ a\ b\ ts \simeq \mathsf{infsup}\ (\mathsf{Nd}\ ts)\ (\mathrm{mod}\ a,b)$

Proof by simultaneous computation induction. The only two nontrivial cases are the ones stemming from the recursion equations for $\mathsf{ab\_sups}$ and $\mathsf{ab\_infs}$. We concentrate on $\mathsf{ab\_infs}$. For succinctness we introduce the following abbreviations:

$$abt \equiv \mathsf{ab\_sup}\ a\ b\ t \qquad abts \equiv \mathsf{ab\_infs}\ a\ (b \sqcap abt)\ ts$$
$$vt \equiv \mathsf{supinf}\ t \qquad vts \equiv \mathsf{infsup}\ (\mathsf{Nd}\ ts)$$

The two IHs are

$$a \sqcup abt \sqcap b = a \sqcup vt \sqcap b \tag{IH1}$$
$$\neg\ b \sqcap abt \leq a \longrightarrow abts \simeq vts\ (\mathrm{mod}\ a,b \sqcap abt) \tag{IH2}$$

and we need to prove

$$\mathsf{ab\_sups}\ a\ b\ (t\ \#\ ts) \simeq \mathsf{supinf}\ (\mathsf{Nd}\ (t\ \#\ ts))\ (\mathrm{mod}\ a,b)$$

The proof is by cases. First we assume $b \sqcap abt \leq a$. Together with IH1 this implies $a \sqcup vt \sqcap b = a$. Now we prove the main equation:

$$\begin{aligned}
&\mathsf{ab\_sups}\ a\ b\ (t\ \#\ ts) \simeq b \sqcap abt\ (\mathrm{mod}\ a,b) && \text{because } b \sqcap abt \leq a\\
&= a \sqcup abt \sqcap b &&\\
&= a \sqcup vt \sqcap b && \text{by IH1}\\
&= a \sqcup vt \sqcap b \sqcup v \sqcap vts \sqcap b &&\\
&= a \sqcup vt \sqcap vts \sqcap b && \text{because } a \sqcup vt \sqcap b = a\\
&= a \sqcup \mathsf{supinf}\ (\mathsf{Nd}\ (t\ \#\ ts)) \sqcap b &&
\end{aligned}$$

Now we assume $\neg\ b \leq a \sqcup abt$. IH2 together with a simple inductive property of $\mathsf{ab\_infs}$, namely $\mathsf{ab\_infs}\ x\ y\ ts \leq y$, implies

$$a \sqcup abts \sqcap b = a \sqcup abt \sqcap vts \sqcap b \tag{IH2'}$$

Now we prove the main equation:

$$\begin{aligned}
&\mathsf{ab\_infs}\ a\ b\ (t\ \#\ ts) \simeq abts\ (\mathrm{mod}\ a,b) && \text{because } \neg\ b \leq a \sqcup abt\\
&= a \sqcup abt \sqcap vts \sqcap b && \text{by IH2'}\\
&= a \sqcup vt \sqcap vts \sqcap b && \text{by IH1}\\
&= (a \sqcup \mathsf{infsup}\ (\mathsf{Nd}\ (t\ \#\ ts))) \sqcap b && \square
\end{aligned}$$

Because $x \simeq y\ (\mathrm{mod}\ \bot,\top)$ implies $x = y$, we obtain:

Corollary 27.8. $\mathsf{ab\_sup}\ \bot\ \top\ t = \mathsf{supinf}\ t$ (27.13)

27.4.4 Negative Values

We can deal with negative values in the context of bounded distributive lattices by requiring the same properties as for De Morgan orders, but with $(\sqcap)$ instead of *min*:

$$-\,(-\,x) = x$$

$$-\,(x \sqcap y) = -\,x \sqcup -\,y$$

The resulting structure is called a **De Morgan algebra**. Just as in Section 27.3 we can define game tree evaluation

```
negsup :: 'a tree ⇒ 'a
negsup (Lf x) = x
negsup (Nd ts) = sups (map (λt. − negsup t) ts)
```

and alpha-beta pruning for De Morgan algebras:

```
ab_negsup :: 'a ⇒ 'a ⇒ 'a tree ⇒ 'a
ab_negsup _ _ (Lf x) = x
ab_negsup a b (Nd ts) = ab_negsups a b ts

ab_negsups :: 'a ⇒ 'a ⇒ 'a tree list ⇒ 'a
ab_negsups a _ [] = a
ab_negsups a b (t # ts)
= (let a' = a ⊔ − ab_negsup (− b) (− a) t
   in if b ≤ a' then a' else ab_negsups a' b ts)
```

We can relate the ordinary and the negated versions

$$\textit{negsup}\ t = \textit{supinf}\ (\textit{negate False}\ t)$$

$$\textit{ab\_sup}\ a\ b\ t = \textit{ab\_negsup}\ a\ b\ (\textit{negate False}\ t)$$

by induction (details omitted, especially the three simultaneous propositions required for the proof of the second proposition) and conclude

$$\textit{ab\_negsup}\ \bot\ \top\ t = \textit{negsup}\ t$$

with the help of (27.13) (and the inductive lemma *negate* f (*negate* f t) $= t$).

27.4.5 Exercises

Exercise 27.8. In Exercise 27.3 we considered a reformulation of "$\le$ mod". This reformulation generalizes to lattices in the standard manner. Define

$$ab \sqsubseteq v \text{ (mod } a,b) \equiv b \sqcap v \le ab \wedge ab \le a \sqcup v$$

It turns out that this is a suitable correctness notion for alpha-beta pruning in distributive lattices. Give a detailed proof of this generalization of Theorem 27.7:

$$\mathit{ab\_sup}\ a\ b\ t \sqsubseteq \mathit{supinf}\ t \text{ (mod } a,b)$$

Obviously $\mathit{ab\_sup}\ \bot\ \top\ t = \mathit{supinf}\ t$ follows immediately.

Give a detailed proof of $ab \sqsubseteq v \text{ (mod } a,b) \longrightarrow ab \simeq v \text{ (mod } a,b)$ and a counterexample to the reverse implication.

Exercise 27.9. There is also a fail-soft version of alpha-beta pruning for distributive lattices:

$$\mathit{ab\_sup'} :: 'a \Rightarrow 'a \Rightarrow 'a\ \mathit{tree} \Rightarrow 'a$$

$$\mathit{ab\_sup'}\ \_\ \_\ (\mathit{Lf}\ x) = x$$
$$\mathit{ab\_sup'}\ a\ b\ (\mathit{Nd}\ ts) = \mathit{ab\_sups'}\ a\ b\ \bot\ ts$$

$$\mathit{ab\_sups'} :: 'a \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a\ \mathit{tree\ list} \Rightarrow 'a$$

```
$ab\_sups'\ \_\ \_\ m\ [] = m$
$ab\_sups'\ a\ b\ m\ (t\ \#\ ts)$
$= ($let $m' = m \sqcup ab\_inf'\ (m \sqcup a)\ b\ t$
    in if $b \le m'$ then $m'$ else $ab\_sups'\ a\ b\ m'\ ts)$
```

Prove its correctness (for "$\sqsubseteq$ mod" see Exercise 27.8):

$$\mathit{ab\_sup'}\ a\ b\ t \sqsubseteq \mathit{supinf}\ t \text{ (mod } a,b)$$
$$\mathit{ab\_sups'}\ a\ b\ m\ ts \sqsubseteq \mathit{supinf}\ (\mathit{Nd}\ ts) \text{ (mod } a \sqcup m,b)$$

Exercise 27.10. Based on the definition of "$\sqsubseteq$ mod" in Exercise 27.8, prove

$$\mathit{ab\_negsup}\ a\ b\ t \sqsubseteq \mathit{negsup}\ t \text{ (mod } a,b)$$
$$\mathit{ab\_negsups}\ a\ b\ ts \sqsubseteq \mathit{negsup}\ (\mathit{Nd}\ ts) \text{ (mod } a,b)$$

directly, i.e. without going back to the non-negative relatives. You may need the lemma $a \le \mathit{ab\_negsups}\ a\ b\ ts$.

Exercise 27.11. The algorithm considered in Exercise 27.6 carries over to distributive lattices, *mutatis mutandis*. Prove

$$a \le b \longrightarrow \mathit{ab\_sup2}\ a\ b\ t = a \sqcup \mathit{supinf}\ t \sqcap b$$

Obviously $\mathit{ab\_sup2}\ \bot\ \top\ t = \mathit{supinf}\ t$ follows immediately.

Chapter Notes

Variants of alpha-beta pruning have a long history in the literature. It appears that the first reasonably precise correctness proof was given by Knuth and Moore [1975] via the relation "$\cong$ mod" (Exercise 27.4). The improvement from fail-hard to fail-soft was proposed by Fishburn [1983] with the suggestion of using it to narrow the a,b window in future searches of the same position. Marsland [1986] spells out the details of the code. Fishburn [1983] contrasts the correctness property "$\cong$ mod" that Knuth and Moore proved of the fail-hard variant with his own stronger correctness property "$\leq$ mod" (27.6) of the fail-soft variant. He does not seem to have realized that fail-hard already satisfies (27.6) and that the distinguishing property is that fail-hard bounds fail-soft (Theorem 27.3).

Hughes [1989] derives a version of alpha-beta pruning for numbers from the definition of *maxmin*. However, he ends up with shallow pruning only, i.e. function $F1$ by Knuth and Moore [1975], not $F2$, the real alpha-beta pruning. In their historic survey, Knuth and Moore [1975, pp.303-304] point out that this mistake has been made frequently, including by Knuth himself.

The fact that alpha-beta pruning extends to distributed lattices was discovered twice. First by Bird and Hughes [1987], who (like Hughes [1989]) derive an algorithm from the definition of *maxmin*. Confusingly they talk about Boolean algebras although they merely work in a distributive lattice. Their version of alpha-beta pruning could be classified as fail-extremely-hard because it always returns a result in the interval $[a,b]$ (see Exercise 27.11). Ginsberg and Jaffray [2002] rediscovered that alpha-beta pruning also works in distributed lattices. Li et al. [2022] extend alpha-beta pruning in distributive lattices to fail-soft on a game graph using a cache. They employ the squashing operation $a \sqcup x \sqcap b$ introduced by Bird and Hughes [1987] to state correctness. Both Ginsberg and Jaffray [2002] and Li et al. [2022] are unaware of the work by Bird and Hughes [1987] who in turn seem unaware of the work by Knuth and Moore [1975].

De Morgan algebras were introduced and studied by Moisil [1936, p. 91] (without the assumption of boundedness). The term "De Morgan order" is not standard and was coined by the author in analogy with De Morgan algebras.

Pearl [1980, 1982] provided the definitive quantitative analysis of alpha-beta pruning and showed that, for random game trees, alpha-beta pruning is optimal.

Part VI

Appendix

A List Library

The following functions on lists are predefined:

$\textit{length} :: {'a}\ \mathit{list} \Rightarrow \mathit{nat}$

$|[]| = 0$
$|x\ \#\ \mathit{xs}| = |\mathit{xs}| + 1$

$(@) :: {'a}\ \mathit{list} \Rightarrow {'a}\ \mathit{list} \Rightarrow {'a}\ \mathit{list}$

$[]\ @\ \mathit{ys} = \mathit{ys}$
$(x\ \#\ \mathit{xs})\ @\ \mathit{ys} = x\ \#\ \mathit{xs}\ @\ \mathit{ys}$

$\textit{set} :: {'a}\ \mathit{list} \Rightarrow {'a}\ \mathit{set}$

$\textit{set}\ [] = \{\}$
$\textit{set}\ (x\ \#\ \mathit{xs}) = \{x\} \cup \textit{set}\ \mathit{xs}$

$\textit{map} :: ({'a} \Rightarrow {'b}) \Rightarrow {'a}\ \mathit{list} \Rightarrow {'b}\ \mathit{list}$

$\textit{map}\ f\ [] = []$
$\textit{map}\ f\ (x\ \#\ \mathit{xs}) = f\ x\ \#\ \textit{map}\ f\ \mathit{xs}$

$\textit{filter} :: ({'a} \Rightarrow \mathit{bool}) \Rightarrow {'a}\ \mathit{list} \Rightarrow {'a}\ \mathit{list}$

$\textit{filter}\ p\ [] = []$
$\textit{filter}\ p\ (x\ \#\ \mathit{xs}) = (\textbf{if}\ p\ x\ \textbf{then}\ x\ \#\ \textit{filter}\ p\ \mathit{xs}\ \textbf{else}\ \textit{filter}\ p\ \mathit{xs})$

$\textit{concat} :: {'a}\ \mathit{list}\ \mathit{list} \Rightarrow {'a}\ \mathit{list}$

$\textit{concat}\ [] = []$
$\textit{concat}\ (x\ \#\ \mathit{xs}) = x\ @\ \textit{concat}\ \mathit{xs}$

$\textit{take} :: \mathit{nat} \Rightarrow {'a}\ \mathit{list} \Rightarrow {'a}\ \mathit{list}$

$\textit{take}\ \_\ [] = []$
$\textit{take}\ n\ (x\ \#\ \mathit{xs}) = (\textbf{case}\ n\ \textbf{of}\ 0 \Rightarrow []\ |\ m + 1 \Rightarrow x\ \#\ \textit{take}\ m\ \mathit{xs})$

```
drop :: nat ⇒ 'a list ⇒ 'a list

drop _ [] = []
drop n (x # xs) = (case n of 0 ⇒ x # xs | m + 1 ⇒ drop m xs)


hd :: 'a list ⇒ 'a

hd (x # xs) = x


tl :: 'a list ⇒ 'a list

tl [] = []
tl (x # xs) = xs


butlast :: 'a list ⇒ 'a list

butlast [] = []
butlast (x # xs) = (if xs = [] then [] else x # butlast xs)


rev :: 'a list ⇒ 'a list

rev [] = []
rev (x # xs) = rev xs @ [x]


(!) :: 'a list ⇒ nat ⇒ 'a

(x # xs) ! n = (case n of 0 ⇒ x | k + 1 ⇒ xs ! k)


list_update :: 'a list ⇒ nat ⇒ 'a ⇒ 'a list

[][_ := _] = []
(x # xs)[i := v] = (case i of 0 ⇒ v # xs | j + 1 ⇒ x # xs[j := v])


upt :: nat ⇒ nat ⇒ nat list

[_..<0] = []
[i..<j + 1] = (if i ≤ j then [i..<j] @ [j] else [])


replicate :: nat ⇒ 'a ⇒ 'a list

replicate 0 _ = []
replicate (n + 1) x = x # replicate n x
```

```
distinct :: 'a list ⇒ bool
distinct [] = True
distinct (x # xs) = (x ∉ set xs ∧ distinct xs)

sum_list :: 'a list ⇒ 'a
sum_list [] = 0
sum_list (x # xs) = x + sum_list xs

min_list :: 'a list ⇒ 'a
min_list (x # xs)
= (case xs of [] ⇒ x | _ # _ ⇒ min x (min_list xs))

sorted_wrt :: ('a ⇒ 'a ⇒ bool) ⇒ 'a list ⇒ bool
sorted_wrt P [] = True
sorted_wrt P (x # ys) = ((∀y∈set ys. P x y) ∧ sorted_wrt P ys)
```

B Time Functions

Time functions that are 0 by definition have already been simplified away.

B.1 Lists

$T_{\mathit{append}} :: {'a}\ \mathit{list} \Rightarrow {'a}\ \mathit{list} \Rightarrow \mathit{nat}$

$T_{\mathit{append}}\ []\ \_ = 1$
$T_{\mathit{append}}\ (\_ \# \mathit{xs})\ \mathit{ys} = T_{\mathit{append}}\ \mathit{xs}\ \mathit{ys} + 1$

$T_{\mathit{length}} :: {'a}\ \mathit{list} \Rightarrow \mathit{nat}$

$T_{\mathit{length}}\ [] = 1$
$T_{\mathit{length}}\ (\_ \# \mathit{xs}) = T_{\mathit{length}}\ \mathit{xs} + 1$

$T_{\mathit{map}} :: ({'a} \Rightarrow \mathit{nat}) \Rightarrow {'a}\ \mathit{list} \Rightarrow \mathit{nat}$

$T_{\mathit{map}}\ \_\ [] = 1$
$T_{\mathit{map}}\ \mathit{Tf}\ (x \# \mathit{xs}) = \mathit{Tf}\ x + T_{\mathit{map}}\ \mathit{Tf}\ \mathit{xs} + 1$

$T_{\mathit{filter}} :: ({'a} \Rightarrow \mathit{nat}) \Rightarrow {'a}\ \mathit{list} \Rightarrow \mathit{nat}$

$T_{\mathit{filter}}\ \mathit{TP}\ [] = 1$
$T_{\mathit{filter}}\ \mathit{TP}\ (x \# \mathit{xs}) = \mathit{TP}\ x + T_{\mathit{filter}}\ \mathit{TP}\ \mathit{xs} + 1$

$T_{\mathit{take}} :: \mathit{nat} \Rightarrow {'a}\ \mathit{list} \Rightarrow \mathit{nat}$

$T_{\mathit{take}}\ \_\ [] = 1$
$T_{\mathit{take}}\ n\ (\_ \# \mathit{xs}) = (\textbf{case}\ n\ \textbf{of}\ 0 \Rightarrow 0 \mid m + 1 \Rightarrow T_{\mathit{take}}\ m\ \mathit{xs}) + 1$

$T_{\mathit{drop}} :: \mathit{nat} \Rightarrow {'a}\ \mathit{list} \Rightarrow \mathit{nat}$

$T_{\mathit{drop}}\ \_\ [] = 1$
$T_{\mathit{drop}}\ n\ (\_ \# \mathit{xs}) = (\textbf{case}\ n\ \textbf{of}\ 0 \Rightarrow 0 \mid m + 1 \Rightarrow T_{\mathit{drop}}\ m\ \mathit{xs}) + 1$

$T_{\mathit{nth}} :: {'a}\ \mathit{list} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat}$

$T_{\mathit{nth}}\ (\_ \# \mathit{xs})\ n = (\textbf{case}\ n\ \textbf{of}\ 0 \Rightarrow 0 \mid x + 1 \Rightarrow T_{\mathit{nth}}\ \mathit{xs}\ x) + 1$

Simple properties:

$T_{\mathit{append}}\ \mathit{xs}\ \mathit{ys} = |\mathit{xs}| + 1$

$T_{\mathit{length}}\ \mathit{xs} = |\mathit{xs}| + 1$

$T_{\mathit{map}}\ \mathit{Tf}\ \mathit{xs} = (\sum_{x \leftarrow \mathit{xs}} \mathit{Tf}\ x) + |\mathit{xs}| + 1$

$T_{\mathit{filter}}\ \mathit{TP}\ \mathit{xs} = (\sum_{x \leftarrow \mathit{xs}} \mathit{TP}\ x) + |\mathit{xs}| + 1$

$T_{\mathit{take}}\ n\ \mathit{xs} = \mathit{min}\ n\ |\mathit{xs}| + 1$

$T_{\mathit{drop}}\ n\ \mathit{xs} = \mathit{min}\ n\ |\mathit{xs}| + 1$

$n < |\mathit{xs}| \longrightarrow T_{\mathit{nth}}\ \mathit{xs}\ n = n + 1$

B.2 Selection

$T_{\mathit{chop}} :: \mathit{nat} \Rightarrow {'a}\ \mathit{list} \Rightarrow \mathit{nat}$

$T_{\mathit{chop}}\ 0\ \_ = 1$
$T_{\mathit{chop}}\ \_\ [] = 1$
$T_{\mathit{chop}}\ n\ \mathit{xs} = T_{\mathit{take}}\ n\ \mathit{xs} + T_{\mathit{drop}}\ n\ \mathit{xs} + T_{\mathit{chop}}\ n\ (\mathit{drop}\ n\ \mathit{xs}) + 1$

$T_{\mathit{partition3}} :: {'a} \Rightarrow {'a}\ \mathit{list} \Rightarrow \mathit{nat}$

$T_{\mathit{partition3}}\ \_\ [] = 1$
$T_{\mathit{partition3}}\ x\ (\_\ \#\ \mathit{ys}) = T_{\mathit{partition3}}\ x\ \mathit{ys} + 1$

$T_{\mathit{slow\_select}} :: \mathit{nat} \Rightarrow {'a}\ \mathit{list} \Rightarrow \mathit{nat}$

$T_{\mathit{slow\_select}}\ k\ \mathit{xs} = T_{\mathit{insort}}\ \mathit{xs} + T_{\mathit{nth}}\ (\mathit{insort}\ \mathit{xs})\ k$

$T_{\mathit{slow\_median}} :: {'a}\ \mathit{list} \Rightarrow \mathit{nat}$

$T_{\mathit{slow\_median}}\ \mathit{xs} = T_{\mathit{length}}\ \mathit{xs} + T_{\mathit{slow\_select}}\ ((|\mathit{xs}| - 1)\ \mathrm{div}\ 2)\ \mathit{xs}$

Simple properties:

$T_{\mathit{chop}}\ d\ \mathit{xs} \le 5 \cdot |\mathit{xs}| + 1$

$T_{\mathit{partition3}}\ x\ \mathit{xs} = |\mathit{xs}| + 1$

$k < |\mathit{xs}| \longrightarrow T_{\mathit{slow\_select}}\ k\ \mathit{xs} \le |\mathit{xs}|^2 + 3 \cdot |\mathit{xs}| + 1$

$\mathit{xs} \neq [] \longrightarrow T_{\mathit{slow\_median}}\ \mathit{xs} \le |\mathit{xs}|^2 + 4 \cdot |\mathit{xs}| + 2$

B.3 2-3 Trees

$T_{\mathit{join\_adj}} :: {'a}\ \mathit{tree23s} \Rightarrow \mathit{nat}$

$T_{\mathit{join\_adj}}\ (\mathit{TTs}\ \_\ \_\ (T\ \_)) = 1$
$T_{\mathit{join\_adj}}\ (\mathit{TTs}\ \_\ \_\ (\mathit{TTs}\ \_\ \_\ (T\ \_))) = 1$
$T_{\mathit{join\_adj}}\ (\mathit{TTs}\ \_\ \_\ (\mathit{TTs}\ \_\ \_\ ts)) = T_{\mathit{join\_adj}}\ ts + 1$

$T_{\mathit{join\_all}} :: {'a}\ \mathit{tree23s} \Rightarrow \mathit{nat}$

$T_{\mathit{join\_all}}\ (T\ \_) = 1$
$T_{\mathit{join\_all}}\ ts = T_{\mathit{join\_adj}}\ ts + T_{\mathit{join\_all}}\ (\mathit{join\_adj}\ ts) + 1$

$T_{\mathit{leaves}} :: {'a}\ \mathit{list} \Rightarrow \mathit{nat}$

$T_{\mathit{leaves}}\ [] = 1$
$T_{\mathit{leaves}}\ (\_\ \#\ as) = T_{\mathit{leaves}}\ as + 1$

$T_{\mathit{tree23\_of\_list}} :: {'a}\ \mathit{list} \Rightarrow \mathit{nat}$

$T_{\mathit{tree23\_of\_list}}\ as = T_{\mathit{leaves}}\ as + T_{\mathit{join\_all}}\ (\mathit{leaves}\ as)$

B.4 Arrays via Braun Trees

$T_{\mathit{nodes}} :: {'a}\ \mathit{tree}\ \mathit{list} \Rightarrow {'a}\ \mathit{list} \Rightarrow {'a}\ \mathit{tree}\ \mathit{list} \Rightarrow \mathit{nat}$

$T_{\mathit{nodes}}\ (\_\ \#\ ls)\ (\_\ \#\ xs)\ (\_\ \#\ rs) = T_{\mathit{nodes}}\ ls\ xs\ rs + 1$
$T_{\mathit{nodes}}\ (\_\ \#\ ls)\ (\_\ \#\ xs)\ [] = T_{\mathit{nodes}}\ ls\ xs\ [] + 1$
$T_{\mathit{nodes}}\ []\ (\_\ \#\ xs)\ (\_\ \#\ rs) = T_{\mathit{nodes}}\ []\ xs\ rs + 1$
$T_{\mathit{nodes}}\ []\ (\_\ \#\ xs)\ [] = T_{\mathit{nodes}}\ []\ xs\ [] + 1$
$T_{\mathit{nodes}}\ \_\ []\ \_ = 1$

B.5 Leftist Heaps

$T_{\mathit{merge}} :: ({'a} \times \mathit{nat})\ \mathit{tree} \Rightarrow ({'a} \times \mathit{nat})\ \mathit{tree} \Rightarrow \mathit{nat}$

$T_{\mathit{merge}}\ \langle\rangle\ \_ = 1$
$T_{\mathit{merge}}\ \_\ \langle\rangle = 1$
$T_{\mathit{merge}}\ (\langle l_1, (a_1, n_1), r_1\rangle =: t_1)\ (\langle l_2, (a_2, n_2), r_2\rangle =: t_2)$

$= (\textbf{if } a_1 \le a_2 \textbf{ then } T_{\mathit{merge}}\ r_1\ t_2 \textbf{ else } T_{\mathit{merge}}\ t_1\ r_2) + 1$

$T_{\mathit{insert}} :: {'a} \Rightarrow ({'a} \times \mathit{nat})\ \mathit{tree} \Rightarrow \mathit{nat}$

$T_{\mathit{insert}}\ x\ t = T_{\mathit{merge}}\ \langle\langle\rangle, (x, 1), \langle\rangle\rangle\ t$

$T_{\mathit{del\_min}} :: ({'a} \times \mathit{nat})\ \mathit{tree} \Rightarrow \mathit{nat}$

$T_{\mathit{del\_min}}\ \langle\rangle = 0$

$T_{\mathit{del\_min}}\ \langle l, \_, r\rangle = T_{\mathit{merge}}\ l\ r$

$T_{\mathit{merge\_all}} :: ({'a} \times \mathit{nat})\ \mathit{tree}\ \mathit{list} \Rightarrow \mathit{nat}$

$T_{\mathit{merge\_all}}\ [] = 0$

$T_{\mathit{merge\_all}}\ [\_] = 0$

$T_{\mathit{merge\_all}}\ \mathit{ts} = T_{\mathit{merge\_all}}\ (\mathit{merge\_adj}\ \mathit{ts}) + T_{\mathit{merge\_adj}}\ \mathit{ts}$

$T_{\mathit{lheap\_list}} :: {'a}\ \mathit{list} \Rightarrow \mathit{nat}$

$T_{\mathit{lheap\_list}}\ \mathit{xs} = T_{\mathit{merge\_all}}\ (\mathit{map}\ (\lambda x.\ \langle\langle\rangle, (x, 1), \langle\rangle\rangle)\ \mathit{xs})$

B.6 Priority Queues Based on Braun Trees

$T_{\mathit{insert}} :: {'a} \Rightarrow {'a}\ \mathit{tree} \Rightarrow \mathit{nat}$

$T_{\mathit{insert}}\ \_\ \langle\rangle = 1$

$T_{\mathit{insert}}\ a\ \langle\_, x, r\rangle = (\textbf{if } a < x \textbf{ then } T_{\mathit{insert}}\ x\ r \textbf{ else } T_{\mathit{insert}}\ a\ r) + 1$

$T_{\mathit{del\_min}} :: {'a}\ \mathit{tree} \Rightarrow \mathit{nat}$

$T_{\mathit{del\_min}}\ \langle\rangle = 0$

$T_{\mathit{del\_min}}\ \langle\langle\rangle, \_, \_\rangle = 0$

$T_{\mathit{del\_min}}\ \langle l, \_, r\rangle = T_{\mathit{del\_left}}\ l + (\textbf{let } (y, l') = \mathit{del\_left}\ l \textbf{ in } T_{\mathit{sift\_down}}\ r\ y\ l')$

$T_{\mathit{del\_left}} :: {'a}\ \mathit{tree} \Rightarrow \mathit{nat}$

$T_{\mathit{del\_left}}\ \langle\langle\rangle, \_, \_\rangle = 1$

$T_{\mathit{del\_left}}\ \langle l, \_, \_\rangle = T_{\mathit{del\_left}}\ l + 1$

$T_{\mathit{sift\_down}} :: {'a}\ \mathit{tree} \Rightarrow {'a} \Rightarrow {'a}\ \mathit{tree} \Rightarrow \mathit{nat}$

$T_{\mathit{sift\_down}}\ \langle\rangle\ \_\ \_ = 1$

$T_{\mathit{sift\_down}}\ \langle\langle\rangle, \_, \_\rangle\ \_\ \langle\rangle = 1$

$T_{\mathit{sift\_down}}\ \langle l_1,\ x_1,\ r_1\rangle\ a\ \langle l_2,\ x_2,\ r_2\rangle$
$= (\textbf{if}\ a \le x_1 \wedge a \le x_2\ \textbf{then}\ 0$
$\quad \textbf{else if}\ x_1 \le x_2\ \textbf{then}\ T_{\mathit{sift\_down}}\ l_1\ a\ r_1\ \textbf{else}\ T_{\mathit{sift\_down}}\ l_2\ a\ r_2) + 1$

B.7 Binomial Priority Queues

The functions T_{link}, T_{rank}, T_{root} and T_{min} are 0 everywhere and have been eliminated from the following definitions.

$T_{\mathit{ins\_tree}} :: {}'a\ \mathit{tree} \Rightarrow {}'a\ \mathit{tree\ list} \Rightarrow \mathit{nat}$

$T_{\mathit{ins\_tree}}\ \_\ [] = 1$
$T_{\mathit{ins\_tree}}\ t_1\ (t_2\ \#\ \mathit{ts})$
$= (\textbf{if}\ \mathit{rank}\ t_1 < \mathit{rank}\ t_2\ \textbf{then}\ 0\ \textbf{else}\ T_{\mathit{ins\_tree}}\ (\mathit{link}\ t_1\ t_2)\ \mathit{ts}) + 1$

$T_{\mathit{insert}} :: {}'a \Rightarrow {}'a\ \mathit{tree\ list} \Rightarrow \mathit{nat}$

$T_{\mathit{insert}}\ x\ \mathit{ts} = T_{\mathit{ins\_tree}}\ (\mathit{Node}\ 0\ x\ [])\ \mathit{ts}$

$T_{\mathit{merge}} :: {}'a\ \mathit{tree\ list} \Rightarrow {}'a\ \mathit{tree\ list} \Rightarrow \mathit{nat}$

$T_{\mathit{merge}}\ \_\ [] = 1$
$T_{\mathit{merge}}\ []\ \_ = 1$
$T_{\mathit{merge}}\ (t_1\ \#\ \mathit{ts}_1 =: h_1)\ (t_2\ \#\ \mathit{ts}_2 =: h_2)$
$= (\textbf{if}\ \mathit{rank}\ t_1 < \mathit{rank}\ t_2\ \textbf{then}\ T_{\mathit{merge}}\ \mathit{ts}_1\ h_2$
$\quad \textbf{else if}\ \mathit{rank}\ t_2 < \mathit{rank}\ t_1\ \textbf{then}\ T_{\mathit{merge}}\ h_1\ \mathit{ts}_2$
$\qquad \textbf{else}\ T_{\mathit{merge}}\ \mathit{ts}_1\ \mathit{ts}_2 + T_{\mathit{ins\_tree}}\ (\mathit{link}\ t_1\ t_2)\ (\mathit{merge}\ \mathit{ts}_1\ \mathit{ts}_2)) + 1$

$T_{\mathit{get\_min}} :: {}'a\ \mathit{tree\ list} \Rightarrow \mathit{nat}$

$T_{\mathit{get\_min}}\ [\_] = 1$
$T_{\mathit{get\_min}}\ (\_\ \#\ \mathit{ts}) = T_{\mathit{get\_min}}\ \mathit{ts} + 1$

$T_{\mathit{get\_min\_rest}} :: {}'a\ \mathit{tree\ list} \Rightarrow \mathit{nat}$

$T_{\mathit{get\_min\_rest}}\ [\_] = 1$
$T_{\mathit{get\_min\_rest}}\ (\_\ \#\ \mathit{ts}) = T_{\mathit{get\_min\_rest}}\ \mathit{ts} + 1$

$T_{\mathit{del\_min}} :: {}'a\ \mathit{tree\ list} \Rightarrow \mathit{nat}$

$T_{\mathit{del\_min}}\ \mathit{ts}$
$= T_{\mathit{get\_min\_rest}}\ \mathit{ts}\ +$

```
  (case get_min_rest ts of
   (Node _ _ $ts_1$, $ts_2$) ⇒ $T_{itrev}$ $ts_1$ [] + $T_{merge}$ (itrev $ts_1$ []) $ts_2$)
```

B.8 Queues

```
$T_{norm}$ :: 'a list × 'a list ⇒ nat
$T_{norm}$ (fs, rs) = (if fs = [] then $T_{itrev}$ rs [] else 0)

$T_{enq}$ :: 'a ⇒ 'a list × 'a list ⇒ nat
$T_{enq}$ a (fs, rs) = $T_{norm}$ (fs, a # rs)

$T_{deq}$ :: 'a list × 'a list ⇒ nat
$T_{deq}$ (fs, rs) = (if fs = [] then 0 else $T_{norm}$ (tl fs, rs))
```

B.9 Splay Trees

```
$T_{splay}$ :: 'a ⇒ 'a tree ⇒ nat
$T_{splay}$ _ ⟨⟩ = 1
$T_{splay}$ x ⟨AB, b, CD⟩
= (case cmp x b of
    LT ⇒ case AB of
           ⟨⟩ ⇒ 0 |
           ⟨A, a, B⟩ ⇒ case cmp x a of
                         LT ⇒ if A = ⟨⟩ then 0 else $T_{splay}$ x A |
                         EQ ⇒ 0 |
                         GT ⇒ if B = ⟨⟩ then 0 else $T_{splay}$ x B |
    EQ ⇒ 0 |
    GT ⇒ case CD of
           ⟨⟩ ⇒ 0 |
           ⟨C, c, D⟩ ⇒ case cmp x c of
                         LT ⇒ if C = ⟨⟩ then 0 else $T_{splay}$ x C |
                         EQ ⇒ 0 |
                         GT ⇒ if D = ⟨⟩ then 0 else $T_{splay}$ x D) + 1
```

```
$T_{splay\_max}$ :: $'a$ tree $\Rightarrow$ nat
$T_{splay\_max}$ $\langle\rangle = 1$
$T_{splay\_max}$ $\langle\_,\ \_,\ \langle\rangle\rangle = 1$
$T_{splay\_max}$ $\langle\_,\ \_,\ \langle\_,\ \_,\ CD\rangle\rangle$
= (if $CD = \langle\rangle$ then 0 else $T_{splay\_max}$ $CD$ + (case splay_max $CD$ of
                                                   $\langle\_,\ \_,\ \_\rangle \Rightarrow 0$)) + 1

$T_{insert}$ :: $'a \Rightarrow\ 'a$ tree $\Rightarrow$ nat
$T_{insert}$ $x$ $t$ = (if $t = \langle\rangle$ then 0 else $T_{splay}$ $x$ $t$)

$T_{delete}$ :: $'a \Rightarrow\ 'a$ tree $\Rightarrow$ nat
$T_{delete}$ $x$ $t$
= (if $t = \langle\rangle$ then 0
    else $T_{splay}$ $x$ $t$ +
         (case splay $x$ $t$ of
          $\langle l,\ a,\ \_\rangle \Rightarrow$
            if $x \neq a$ then 0
            else if $l = \langle\rangle$ then 0
                 else $T_{splay\_max}$ $l$ + (case splay_max $l$ of $\langle\_,\ \_,\ \_\rangle \Rightarrow 0$)))
```

B.10 Skew Heaps

```
$T_{merge}$ :: $'a$ tree $\Rightarrow\ 'a$ tree $\Rightarrow$ nat
$T_{merge}$ $\langle\rangle$ $\_$ = 1
$T_{merge}$ $\_$ $\langle\rangle$ = 1
$T_{merge}$ $\langle l_1,\ a_1,\ r_1\rangle$ $\langle l_2,\ a_2,\ r_2\rangle$
= (if $a_1 \le a_2$ then $T_{merge}$ $\langle l_2,\ a_2,\ r_2\rangle$ $r_1$ else $T_{merge}$ $\langle l_1,\ a_1,\ r_1\rangle$ $r_2$) + 1

$T_{insert}$ :: $'a \Rightarrow\ 'a$ tree $\Rightarrow$ int
$T_{insert}$ $a$ $t$ = $T_{merge}$ $\langle\langle\rangle,\ a,\ \langle\rangle\rangle$ $t$

$T_{del\_min}$ :: $'a$ tree $\Rightarrow$ int
$T_{del\_min}$ $t$ = (case $t$ of $\langle\rangle \Rightarrow 0$ | $\langle t_1,\ \_,\ t_2\rangle \Rightarrow$ $T_{merge}$ $t_1$ $t_2$)
```

B.11 Pairing Heaps

The functions T_{link} and T_{merge} are 0 everywhere and have been eliminated from the following definitions.

$T_{insert} :: {'a} \Rightarrow {'a}\ hp\ option \Rightarrow nat$

$T_{insert}\ \_\ \ None = 0$
$T_{insert}\ \_\ \ (Some\ \_) = 0$

$T_{del\_min} :: {'a}\ hp\ option \Rightarrow nat$

$T_{del\_min}\ None = 0$
$T_{del\_min}\ (Some\ (Hp\ \_\ \ hs)) = T_{pass_1}\ hs + T_{pass_2}\ (pass_1\ hs)$

$T_{pass_1} :: {'a}\ hp\ list \Rightarrow nat$

$T_{pass_1}\ (\_\ \#\ \_\ \#\ hs) = T_{pass_1}\ hs + 1$
$T_{pass_1}\ \_\ = 1$

$T_{pass_2} :: {'a}\ hp\ list \Rightarrow nat$

$T_{pass_2}\ [] = 1$
$T_{pass_2}\ (\_\ \#\ hs) = T_{pass_2}\ hs + (\textbf{case}\ pass_2\ hs\ \textbf{of}\ None \Rightarrow 0 \mid \_ \Rightarrow 0) + 1$

C Notation

C.1 Symbol Table

The following table gives an overview of all the special symbols used in this book and how to enter them into Isabelle. The second column shows the full internal name of the symbol; the third column shows additional ASCII abbreviations. Either of these can be used to input the character using the auto-completion popup.

	Code	ASCII abbrev.	Comment
λ	`\<lambda>`	`%`	function abstraction
≡	`\<equiv>`	`==`	meta equality
≠	`\<noteq>`	`~=`	
⋀	`\<And>`	`!!`	meta ∀-quantifier
∀	`\<forall>`	`!`	HOL ∀-quantifier
∃	`\<exists>`	`?`	
⟹	`\<Longrightarrow>`	`==>`	meta implication
⟶	`\<longrightarrow>`	`->`	HOL implication
⟷	`\<longleftrightarrow>`	`<->` or `<-->`	
⇒	`\<Rightarrow>`	`=>`	arrow in function types
←	`\<leftarrow>`	`<-`	list comprehension syntax
¬	`\<not>`	`~`	
∧	`\<and>`	`/\` or `&`	
∨	`\<or>`	`\/` or `|`	
∈	`\<in>`	`:`	
∉	`\<notin>`	`~:`	
∪	`\<union>`	`Un`	
∩	`\<inter>`	`Int`	
⋃	`\<Union>`	`Union` or `UN`	union/intersection of a set of sets
⋂	`\<Inter>`	`Inter` or `INT`	
⊆	`\<subseteq>`	`(=`	
⊂	`\<subset>`		
≤	`\<le>`	`<=`	

	Code	ASCII abbrev.	Comment
≥	`\<ge>`	`>=`	
∘	`\<circ>`		function composition
×	`\<times>`	`<*>`	cartesian prod., prod. type
|	`\<bar>`	`||`	absolute value
⌊	`\<lfloor>`	`[.`	*floor*
⌋	`\<rfloor>`	`.]`	
⌈	`\<lceil>`	`[.`	*ceiling*
⌉	`\<rceil>`	`.]`	
∑	`\<Sum>`	`SUM`	see Section C.3
∏	`\<Prod>`	`PROD`	

Note that the symbols "⦃" and "⦄" that are used in the notation for multisets in this book do not exist in Isabelle; instead, the ASCII notation `{#` and `#}` is used (cf. Section C.3).

C.2 Subscripts and Superscripts

In addition to this, subscripts and superscripts with a single symbol can be rendered using two special symbols, `\<^sub>` and `\<^sup>`. The term x_0 for instance can be input as `x\<^sub>0`.

Longer subscripts and superscripts can be written using the symbols `\<^bsub>`...`\<^esub>` and `\<^bsup>`...`\<^esup>`, but this is only rendered in the somewhat visually displeasing form ⇘...⇙ and ⇗...⇖ by Isabelle/jEdit.

C.3 Syntactic Sugar

The following table lists relevant syntactic sugar that is used in the book or its supplementary material. In some cases, the book notation deviates slightly from the Isabelle notation for better readability.

The last column gives the formal meaning of the notation (i.e. what it expands to). In most cases, this is not important for the user to know, but it can occasionally be useful to find relevant lemmas, or to understand that e.g. if one encounters the term *sum* $f\ A$, this is just the η-contracted form of $\sum x{\in}A.\ f\ x$.

The variables in the table follow the following convention:

- x and y are of arbitrary type
- m and n are natural numbers
- P and Q are Boolean values or predicates
- xs is a list
- A is a set
- M is a multiset

Book notation	Isabelle notation	Internal form
	Arithmetic (for numeric types)	
$x \cdot y$	$x * y$	*times* $x\ y$
$x\ /\ y$ or $\frac{x}{y}$	$x\ /\ y$	*divide* $x\ y$ (for type *real*)
x div y	x div y	*divide* $x\ y$ (for type *nat* or *int*)
$\lvert x\rvert$	$\lvert x\rvert$	*abs* x
$\lfloor x\rfloor$	$\lfloor x\rfloor$	*floor* x
$\lceil x\rceil$	$\lceil x\rceil$	*ceiling* x
x^n	$x\ \hat{}\ n$	*power* $x\ n$

Book notation	Isabelle notation	Internal form
	Lists	
$\lvert xs \rvert$		*length* xs
$[]$	$[]$	*Nil*
$x \# xs$	$x \# xs$	*Cons* $x\ xs$
$[x, y]$	$[x, y]$	$x \# y \# []$
$[m..{<}n]$	$[m..{<}n]$	*upt* m n
$xs\ !\ n$	$xs\ !\ n$	*nth* $xs\ n$
$xs[n := y]$	$xs[n := y]$	*list_update* $xs\ n\ y$
	Sets	
$\{\}$	$\{\}$	*empty*
$\{x, y\}$	$\{x, y\}$	*insert* x (*insert* y $\{\}$)
$x \in A$	$x \in A$	*member* $x\ A$
$x \notin A$	$x \notin A$	$\neg(x \in A)$
$A \cup B$	$A \cup B$	*union* $A\ B$
$A \cap B$	$A \cap B$	*inter* $A\ B$
$A \subseteq B$	$A \subseteq B$	*subset_eq* $A\ B$
$A \subset B$	$A \subset B$	*subset* $A\ B$
f ‘ A	f ‘ A	*image* $f\ A$
$\{x \mid P\ x\}$	$\{x.\ P\ x\}$	*Collect* P
$\{x \in A \mid P\ x\}$	$\{x{\in}A.\ P\ x\}$	$\{x.\ P\ x \wedge x \in A\}$
$\{f\ x\ y \mid P\ x\ y\}$	$\{f\ x\ y\ \mid x\ y.\ P\ x\ y\}$	$\{z.\ \exists x\ y.\ z = f\ x\ y \wedge P\ x\ y\}$
$\bigcup_{x \in A} f\ x$	$\bigcup x{\in}A.\ f\ x$	$\bigcup(f$ ‘ $A)$
$\forall x{\in}A.\ P\ x$	$\forall x{\in}A.\ P\ x$	*Ball* $A\ P$
$\exists x{\in}A.\ P\ x$	$\exists x{\in}A.\ P\ x$	*Bex* $A\ P$

Book notation	Isabelle notation	Internal form
	Multisets	
$\lvert M \rvert$		*size* M
$\{\!\{\}\!\}$	$\{\#\}$	$0 :: {}'a\ \mathit{multiset}$
$\{\!\{x,\ y\}\!\}$	$\{\#x,\ y\#\}$	*add_mset* x (*add_mset* y $\{\#\}$)
$x \in_{\#} M$	$x \in\!\# \ M$	$x \in$ *set_mset* M
$x \notin_{\#} M$	$x \notin\!\# \ M$	$\neg(x \in\!\# \ M)$
$\{\!\{x \in_{\#} M \mid P\ x\}\!\}$	$\{\#\ x{\in}\#\ M.\ P\ x\ \#\}$	*filter_mset* $P\ M$
$\{\!\{f\ x \mid x \in_{\#} M\}\!\}$	$\{\#\ f\ x.\ x \in\!\#\ M\ \#\}$	*image_mset* $f\ M$
$\forall x{\in_{\#}}M.\ P\ x$	$\forall x{\in}\#M.\ P\ x$	$\forall x{\in}$*set_mset* $M.\ P\ x$
$\exists x{\in_{\#}}M.\ P\ x$	$\exists x{\in}\#M.\ P\ x$	$\exists x{\in}$*set_mset* $M.\ P\ x$
$M \subseteq_{\#} M'$	$M \subseteq\!\#\ M'$	*subseteq_mset* $M\ M'$
	Sums	
$\sum A$	$\sum A$	*sum* $(\lambda x.\ x)\ A$
$\sum_{x \in A} f\ x$	$\sum x{\in}A.\ f\ x$	*sum* $f\ A$
$\sum_{k\ =\ i}^{j} f\ k$	$\sum k{=}i..j.\ f\ k$	*sum* $f\ \{i..j\}$
$\sum_{\#} M$	$\sum_{\#}M$	*sum_mset* M
$\sum_{x \in_{\#} M} f\ x$	$\sum x{\in}\#M.\ f\ x$	*sum_mset* (*image_mset* $f\ M$)
$\sum_{x \leftarrow xs} f\ x$	$\sum x{\leftarrow}xs.\ f\ x$	*sum_list* (*map* $f\ xs$)
	(analogous for products)	
	Intervals (for ordered types)	
$\{x..\}$	$\{x..\}$	*atLeast* x
$\{..y\}$	$\{..y\}$	*atLeast* y
$\{x..y\}$	$\{x..y\}$	*atLeastAtMost* $x\ y$
$\{x..{<}y\}$	$\{x..{<}y\}$	*atLeastLessThan* $x\ y$
$\{x{<}..y\}$	$\{x{<}..y\}$	*greaterThanAtMost* $x\ y$
$\{x{<}..{<}y\}$	$\{x{<}..{<}y\}$	*greaterThanLessThan* $x\ y$

Bibliography

M. Abdulaziz, K. Mehlhorn, and T. Nipkow. 2019. Trustworthy graph algorithms (invited paper). In *The 44th International Symposium on Mathematical Foundations of Computer Science (MFCS)*, volume 138, pp. 1:1–1:22. DOI: 10.4230/LIPIcs.MFCS.2019.1.

S. Adams. 1993. Efficient aets—A balancing act. *J. Funct. Program.*, 3(4): 553–561. https://doi.org/10.1017/S0956796800000885.

G. M. Adel’son-Vel’skiĭ and E. M. Landis. 1962. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3: 1259–1263.

D. Aingworth, C. Chekuri, P. Indyk, and R. Motwani. 1999. Fast Estimation of Diameter and Shortest Paths (Without Matrix Multiplication). *SIAM Journal on Computing*, 28(4): 1167–1181. DOI: 10.1137/S0097539796303421.

M. Akra and L. Bazzi. 1998. On the solution of linear recurrence equations. *Computational Optimization and Applications*, 10(2): 195–210. https://doi.org/10.1023/A:1018373005182.

S. Aluru. 2017. Quadtrees and octrees. In D. P. Mehta and S. Sahni, eds., *Handbook of Data Structures and Applications*. Chapman and Hall/CRC, 2nd. https://doi.org/10.1201/9781315119335.

A. Appel, 2011. Efficient verified red-black trees. https://www.cs.princeton.edu/~appel/papers/redblack.pdf.

A. W. Appel and X. Leroy. 2023. Efficient extensional binary tries. *J. Autom. Reason.*, 67(1): 8. https://doi.org/10.1007/s10817-022-09655-x.

C. Ballarin. *Tutorial to Locales and Locale Interpretation*. https://isabelle.in.tum.de/doc/locales.pdf.

R. Bayer. 1972. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1: 290–306. DOI: https://doi.org/10.1007/BF00289509.

J. L. Bentley. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9): 509517. https://doi.org/10.1145/361002.361007.

J. L. Bentley and R. Sedgewick. 1997. Fast algorithms for sorting and searching strings. In M. E. Saks, ed., *Symposium on Discrete Algorithms*, pp. 360–369. ACM/SIAM. https://dl.acm.org/doi/10.5555/314161.314321.

S. Berghofer and T. Nipkow. 2002. Executing Higher Order Logic. In *Types for Proofs and Programs*, pp. 24–40. Berlin, Heidelberg. DOI: 10.1007/3-540-45842-5_2.

S. Berghofer and M. Wenzel. 1999. Inductive datatypes in HOL - lessons learned in formal-logic engineering. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin-Mohring, and L. Théry, eds., *Theorem Proving in Higher Order Logics, TPHOLs’99*, volume 1690 of *LNCS*, pp. 19–36. Springer. https://doi.org/10.1007/3-540-48256-3_3.

R. S. Bird and J. Hughes. 1987. The alpha-beta algorithm: An exercise in program transformation. *Inf. Process. Lett.*, 24(1): 53–57. https://doi.org/10.1016/0020-0190(87)90198-0.

J. C. Blanchette. 2009. Proof pearl: Mechanizing the textbook proof of Huffman's algorithm in Isabelle/HOL. *J. Autom. Reason.*, 43(1): 1–18. https://doi.org/10.1007/s10817-009-9116-y.

G. E. Blelloch, D. Ferizovic, and Y. Sun. 2022. Joinable parallel balanced binary trees. *ACM Trans. Parallel Comput.*, 9(2): 7:1–7:41. https://doi.org/10.1145/3512769.

M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan. 1973. Time bounds for selection. *J. Comput. Syst. Sci*, 7(4): 448–461. https://doi.org/10.1016/S0022-0000(73)80033-9.

F. W. Burton. 1982. An efficient functional implementation of FIFO queues. *Inf. Process. Lett.*, 14(5): 205–206. https://doi.org/10.1016/0020-0190(82)90015-1.

S. Cho and S. Sahni. 1998. Weight-biased leftist trees and modified skip lists. *ACM J. Exp. Algorithmics*, 3: 2. https://doi.org/10.1145/297096.297111.

T.-R. Chuang and B. Goldberg. 1993. Real-time deques, multihead Turing machines, and purely functional programming. In *Functional programming languages and computer architecture - FPCA '93*, pp. 289–298. ACM. https://doi.org/10.1145/165180.165225.

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. 2009. *Introduction to Algorithms, 3rd Edition*. MIT Press.

C. A. Crane. 1972. *Linear Lists and Priority Queues as Balanced Binary Trees*. PhD thesis, Stanford University. STAN-CS-72-259.

K. Culík II and D. Wood. 1982. A note on some tree similarity measures. *Inf. Process. Lett.*, 15(1): 39–42. https://doi.org/10.1016/0020-0190(82)90083-7.

R. De La Briandais. 1959. File searching using variable length keys. In *Western Joint Computer Conference*, IRE-AIEE-ACM '59 (Western), pp. 295–298. ACM. http://doi.acm.org/10.1145/1457838.1457895.

M. Eberl. 2017a. The number of comparisons in quicksort. *Archive of Formal Proofs*. http://isa-afp.org/entries/Quick_Sort_Cost.html, Formal proof development.

M. Eberl. 2017b. Proving divide and conquer complexities in Isabelle/HOL. *J. Autom. Reason.*, 58(4): 483–508. https://doi.org/10.1007/s10817-016-9378-0.

M. Eberl, M. W. Haslbeck, and T. Nipkow. 2018. Verified analysis of random binary tree structures. In J. Avigad and A. Mahboubi, eds., *Interactive Theorem Proving (ITP 2018)*, volume 10895 of *LNCS*, pp. 196–214. Springer. https://doi.org/10.1007/978-3-319-94821-8_12.

J. Filliâtre and P. Letouzey. 2004. Functors for proofs and programs. In D. A. Schmidt, ed., *Programming Languages and Systems, ESOP 2004*, volume 2986 of *LNCS*, pp. 370–384. Springer. https://doi.org/10.1007/978-3-540-24725-8_26.

J. P. Fishburn. 1983. Another optimization of alpha-beta search. *SIGART Newsl.*, 84: 37–38. https://doi.org/10.1145/1056623.1056628.

E. Fredkin. 1960. Trie memory. *Commun. ACM*, 3(9): 490–499. https://doi.org/10.1145/367390.367400.

M. L. Fredman, R. Sedgewick, D. Sleator, and R. Tarjan. 1986. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1): 111–129. https://doi.org/10.1007/BF01840439.

J. H. Friedman, J. L. Bentley, and R. A. Finkel. 1977. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3(3): 209226. https://doi.org/10.1145/355744.355745.

K. Germane and M. Might. 2014. Deletion: The curse of the red-black tree. *J. Funct. Program.*, 24(4): 423–433. https://doi.org/10.1017/S0956796814000227.

M. L. Ginsberg and A. Jaffray. 2002. Alpha-beta pruning under partial orders. In R. J. Nowakowski, ed., *More Games of No Chance*, volume 42 of *MSRI Publications*, pp. 37–48. http://library.msri.org/books/Book42/files/ginsberg.pdf.

D. Greenaway, J. Andronick, and G. Klein. 2012. Bridging the Gap: Automatic Verified Abstraction of C. In *Interactive Theorem Proving*, pp. 99–115. Berlin, Heidelberg. DOI: 10.1007/978-3-642-32347-8_8.

L. J. Guibas and R. Sedgewick. 1978. A dichromatic framework for balanced trees. In *Symposium on Foundations of Computer Science (FOCS)*, pp. 8–21. https://doi.org/10.1109/SFCS.1978.3.

F. Haftmann. a. *Haskell-style type classes with Isabelle/Isar*. http://isabelle.in.tum.de/doc/classes.pdf.

F. Haftmann. b. *Code generation from Isabelle/HOL theories*. http://isabelle.in.tum.de/doc/codegen.pdf.

F. Haftmann and T. Nipkow. 2010. Code generation via higher-order rewrite systems. In M. Blume, N. Kobayashi, and G. Vidal, eds., *Functional and Logic Programming (FLOPS 2010)*, volume 6009 of *LNCS*, pp. 103–117. Springer. https://doi.org/10.1007/978-3-642-12251-4_9.

Haskell. Haskell website. https://www.haskell.org.

R. Hinze. 2018. On constructing 2-3 trees. *J. Funct. Program.*, 28: e19. https://doi.org/10.1017/S0956796818000187.

C. A. R. Hoare. 1961. Algorithm 65: Find. *Commun. ACM*, 4(7): 321–322. https://doi.org/10.1145/366622.366647.

C. M. Hoffmann and M. J. O'Donnell. 1982. Programming with equations. *ACM Trans. Program. Lang. Syst.*, 4(1): 83–112. https://doi.org/10.1145/357153.357158.

R. Hood. 1982. *The Efficient Implementation of Very-high-level Programming Language Constructs*. PhD thesis, Department of Computer Science, Cornell University. https://hdl.handle.net/1813/6343.

R. Hood and R. Melville. 1981. Real-time queue operation in pure LISP. *Inf. Process. Lett.*, 13(2): 50–54. https://doi.org/10.1016/0020-0190(81)90030-2.

R. R. Hoogerwoord. 1992. A logarithmic implementation of flexible arrays. In R. Bird, C. Morgan, and J. Woodcock, eds., *Mathematics of Program Construction*, volume 669 of *LNCS*, pp. 191–207. Springer. https://doi.org/10.1007/3-540-56625-2_14.

J. E. Hopcroft and R. M. Karp. 1973. An $n^{5/2}$ Algorithm for Maximum Matchings in Bipartite Graphs. *SIAM J. Comput.*, 2(4): 225–231. DOI: 10.1137/0202019.

B. Huffman and O. Kuncar. 2013. Lifting and Transfer: A Modular Design for Quotients in Isabelle/HOL. In *Certified Programs and Proofs - Third International Conference, CPP 2013, Melbourne, VIC, Australia, December 11-13, 2013, Proceedings*, volume 8307, pp. 131–146. DOI: 10.1007/978-3-319-03545-1_9.

D. A. Huffman. 1952. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9): 1098–1101. https://doi.org/10.1109/JRPROC.1952.273898.

J. Hughes. 1989. Why Functional Programming Matters. *The Computer Journal*, 32(2): 98–107. https://doi.org/10.1093/comjnl/32.2.98.

J. Iacono. 2000. Improved upper bounds for pairing heaps. In M. M. Halldórsson, ed., *Algorithm Theory - SWAT 2000*, volume 1851 of *LNCS*, pp. 32–45. Springer. https://doi.org/10.1007/3-540-44985-X_5.

J. Iacono and M. V. Yagnatinsky. 2016. A linear potential function for pairing heaps. In T. H. Chan, M. Li, and L. Wang, eds., *Combinatorial Optimization and Applications, COCOA 2016*, volume 10043 of *LNCS*, pp. 489–504. Springer. https://doi.org/10.1007/978-3-319-48749-6_36.

C. B. Jones. 1990. *Systematic Software Development using VDM*, 2nd. Prentice Hall International.

S. Kahrs. 2001. Red black trees with types. *J. Funct. Program.*, 11(4): 425–432. https://doi.org/10.1017/S0956796801004026.

A. Kaldewaij and B. Schoenmakers. 1991. The derivation of a tighter bound for top-down skew heaps. *Inf. Process. Lett.*, 37: 265–271. https://doi.org/10.1016/0020-0190(91)90218-7.

Kanellakis. ACM Paris Kanellakis Theory and Practice Award. https://awards.acm.org/kanellakis.

R. M. Karp. 1994. Probabilistic recurrence relations. *J. ACM*, 41(6): 1136–1150. https://doi.org/10.1145/195613.195632.

D. J. King. 1994. Functional binomial queues. In K. Hammond, D. N. Turner, and P. M. Sansom, eds., *Glasgow Workshop on Functional Programming*, Workshops in Computing, pp. 141–150. Springer. https://doi.org/10.1007/978-1-4471-3573-9_10.

D. E. Knuth. 1971. Optimum binary search trees. *Acta Informatica*, 1: 14–25. https://doi.org/10.1007/BF00264289.

D. E. Knuth. 1982. Huffman's algorithm via algebra. *J. Comb. Theory, Ser. A*, 32(2): 216–224. https://doi.org/10.1016/0097-3165(82)90021-8.

D. E. Knuth. 1997. *The Art of Computer Programming, vol. 1: Fundamental Algorithms*, 3rd. Addison–Wesley.

D. E. Knuth and R. W. Moore. 1975. An analysis of alpha-beta pruning. *Artif. Intell.*, 6(4): 293–326. https://doi.org/10.1016/0004-3702(75)90019-3.

D. E. Knuth, J. H. Morris, Jr., and V. R. Pratt. 1977. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2): 323–350.

A. Krauss. *Defining Recursive Functions in Isabelle/HOL*. http://isabelle.in.tum.de/doc/functions.pdf.

A. Krauss. 2006. Partial recursive functions in higher-order logic. In U. Furbach and N. Shankar, eds., *Automated Reasoning,IJCAR 2006*, volume 4130 of *LNCS*, pp. 589–603. Springer. https://doi.org/10.1007/11814771_48.

P. Lammich. November 2009. Collections framework. *Archive of Formal Proofs*. https://isa-afp.org/entries/Collections.html, Formal proof development.

P. Lammich. 2019. Refinement to Imperative HOL. *Journal of Automated Reasoning*, 62(4): 481–503. DOI: 10.1007/s10817-017-9437-1.

P. Lammich and T. Nipkow. 2019. Proof Pearl: Purely Functional, Simple and Efficient Priority Search Trees and Applications to Prim and Dijkstra. In J. Harrison, J. O'Leary, and A. Tolmach, eds., *Interactive Theorem Proving (ITP 2019)*, volume 141 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pp. 23:1–23:18. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.ITP.2019.23.

P. Lammich and S. R. Sefidgar. 2019. Formalizing Network Flow Algorithms: A Refinement Approach in Isabelle/HOL. *J. Autom. Reason.*, 62(2): 261–280. DOI: 10.1007/s10817-017-9442-4.

P. Lammich and T. Tuerk. 2012. Applying Data Refinement for Monadic Programs to Hopcroft's Algorithm. In *Interactive Theorem Proving*, pp. 166–182. Berlin, Heidelberg. DOI: 10.1007/978-3-642-32347-8_12.

D. H. Larkin, S. Sen, and R. E. Tarjan. 2014. A back-to-basics empirical study of priority queues. In C. C. McGeoch and U. Meyer, eds., *2014 Proceedings of the Meeting on Algorithm Engineering and Experiments, ALENEX 2014*, pp. 61–72. SIAM. https://doi.org/10.1137/1.9781611973198.7.

T. Leighton, 1996. Notes on better master theorems for divide-and-conquer recurrences. Lecture notes, MIT. https://courses.csail.mit.edu/6.046/spring04/handouts/akrabazzi.pdf.

J. Li, B. Zanuttini, T. Cazenave, and V. Ventos. 2022. Generalisation of alpha-beta search for AND-OR graphs with partially ordered values. In L. D. Raedt, ed., *International Joint Conference on Artificial Intelligence, IJCAI 2022*, pp. 4769–4775. ijcai.org. https://doi.org/10.24963/ijcai.2022/661.

T. A. Marsland. 1986. A review of game-tree pruning. *J. Int. Comput. Games Assoc.*, 9(1): 3–19. https://doi.org/10.3233/ICG-1986-9102.

D. Meagher. 1982. Geometric modeling using octree encoding. *Comput. Graph. Image Process.*, 19(2): 129–147. https://doi.org/10.1016/0146-664X(82)90104-6.

R. Meis, F. Nielsen, and P. Lammich. 2010. Binomial heaps and skew binomial heaps. *Archive of Formal Proofs*. http://isa-afp.org/entries/Binomial-Heaps.html, Formal proof development.

G. C. Moisil. 1936. Recherches sur l'algèbre de la logique. *Annales scientifiques de l'Université de Jassy*, 122: 1118.

D. R. Morrison. 1968. PATRICIA - practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4): 514–534. https://doi.org/10.1145/321479.321481.

P. Müller. 2018. The binomial heap verification challenge in Viper. In P. Müller and I. Schaefer, eds., *Principled Software Development*, pp. 203–219. Springer. https://doi.org/10.1007/978-3-319-98047-8_13.

D. R. Musser. 1997. Introspective sorting and selection algorithms. *Software: Practice and Experience*, 27(8): 983–993. https://doi.org/10.1002/(SICI)1097-024X(199708)27%3A8%3C983%3A%3AAID-SPE117%3E3.0.CO%3B2-%23.

T. Nipkow. *Programming and Proving in Isabelle/HOL*. http://isabelle.in.tum.de/doc/prog-prove.pdf.

T. Nipkow. 2015. Amortized complexity verified. In C. Urban and X. Zhang, eds., *Interactive Theorem Proving (ITP 2015)*, volume 9236 of *LNCS*, pp. 310–324. Springer. https://doi.org/10.1007/978-3-319-22102-1_21.

T. Nipkow. 2016. Automatic functional correctness proofs for functional search trees. In J. Blanchette and S. Merz, eds., *Interactive Theorem Proving (ITP 2016)*, volume 9807 of *LNCS*, pp. 307–322. Springer. https://doi.org/10.1007/978-3-319-43144-4_19.

T. Nipkow and H. Brinkop. 2019. Amortized complexity verified. *J. Autom. Reason.*, 62(3): 367–391. https://doi.org/10.1007/s10817-018-9459-3.

T. Nipkow and G. Klein. 2014. *Concrete Semantics with Isabelle/HOL*. Springer. http://concrete-semantics.org.

T. Nipkow and T. Sewell. 2020. Proof pearl: Braun trees. In J. Blanchette and C. Hritcu, eds., *Certified Programs and Proofs, CPP 2020*, pp. 18–31. ACM. https://doi.org/10.1145/3372885.3373834.

T. Nipkow and D. Somogyi. 2018. Optimal binary search trees. *Archive of Formal Proofs*. https://isa-afp.org/entries/Optimal_BST.html, Formal proof development.

T. Nipkow, L. Paulson, and M. Wenzel. 2002. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer.

L. Noschinski. 2015. A Graph Library for Isabelle. *Math. Comput. Sci.*, 9(1): 23–39. DOI: 10.1007/S11786-014-0183-Z.

OCaml. OCaml website. https://ocaml.org.

C. Okasaki. 1997. Three algorithms on Braun trees. *J. Funct. Program.*, 7(6): 661–666. https://doi.org/10.1017/s0956796897002876.

C. Okasaki. 1998. *Purely Functional Data Structures*. Cambridge University Press.

L. C. Paulson. 1989. The foundation of a generic theorem prover. *J. Autom. Reason.*, 5: 363–397.

L. C. Paulson. 1996. *ML for the Working Programmer*, 2nd. Cambridge University Press.

J. Pearl. 1980. Asymptotic properties of minimax trees and game-searching procedures. *Artif. Intell.*, 14(2): 113–138. https://doi.org/10.1016/0004-3702(80)90037-5.

J. Pearl. 1982. The solution for the branching factor of the alpha-beta pruning algorithm and its optimality. *Commun. ACM*, 25(8): 559–564. https://doi.org/10.1145/358589.358616.

S. Pettie. 2005. Towards a final analysis of pairing heaps. In *Symposium on Foundations of Computer Science (FOCS)*, pp. 174–183. IEEE Computer Society. https://doi.org/10.1109/SFCS.2005.75.

F. Pottier, A. Guéneau, J.-H. Jourdan, and G. Mével. jan 2024. Thunks and debits in separation logic with time credits. *Proc. ACM Program. Lang.*, 8(POPL). https://doi.org/10.1145/3632892.

L. Pournin. 2014. The diameter of associahedra. *Advances in Mathematics*, 259: 13–42. https://www.sciencedirect.com/science/article/pii/S0001870814000978.

M. Rau. May 2019. Multidimensional binary search trees. *Archive of Formal Proofs*. https://isa-afp.org/entries/KD_Tree.html, Formal proof development.

C. Reade. 1992. Balanced trees with removals: An exercise in rewriting and proof. *Sci. Comput. Program.*, 18(2): 181–204. https://doi.org/10.1016/0167-6423(92)90009-Z.

M. Rem and W. Braun, 1983. A logarithmic implementation of flexible arrays. Memorandum MR83/4. Eindhoven University of Techology.

H. Samet. 1984. The quadtree and related hierarchical data structures. *ACM Comput. Surv.*, 16(2): 187–260. https://doi.org/10.1145/356924.356930.

H. Samet. 1990. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley.

D. Sands. 1990. *Calculi for time analysis of functional programs*. PhD thesis, Imperial College London. http://hdl.handle.net/10044/1/46536.

D. Sands. 1995. A naïve time analysis and its theory of cost equivalence. *J. Log. Comput.*, 5(4): 495–541. https://doi.org/10.1093/logcom/5.4.495.

B. Schoenmakers. 1993. A systematic analysis of splaying. *Inf. Process. Lett.*, 45: 41–50. https://doi.org/10.1016/0020-0190(93)90249-9.

D. D. Sleator and R. E. Tarjan. 1985. Self-adjusting binary search trees. *J. ACM*, 32(3): 652–686. https://doi.org/10.1145/3828.3835.

D. D. Sleator and R. E. Tarjan. 1986. Self-adjusting heaps. *SIAM J. Comput.*, 15(1): 52–69. https://doi.org/10.1137/0215004.

D. D. Sleator, R. E. Tarjan, and W. P. Thurston. 1986. Rotation distance, triangulations, and hyperbolic geometry. In J. Hartmanis, ed., *Symposium on Theory of Computing, 1986*, pp. 122–135. ACM. https://doi.org/10.1145/12130.12143.

R. E. Tarjan. 1985. Amortized computational complexity. *SIAM J. Alg. Disc. Meth.*, 6(2): 306–318. https://doi.org/10.1137/0606031.

L. Théry. 2004. Formalising Huffman's algorithm. Technical Report TRCS 034, Department of Informatics, University of L'Aquila. https://hal.science/hal-02149909/document.

B. Tóth and T. Nipkow. 2023. Real-time double-ended queue verified (proof pearl). In A. Naumowicz and R. Thiemann, eds., *Interactive Theorem Proving, ITP 2023*, volume 268 of *LIPIcs*, pp. 29:1–29:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.ITP.2023.29.

J. Vuillemin. 1978. A data structure for manipulating priority queues. *Commun. ACM*, 21(4): 309–315. https://doi.org/10.1145/359460.359478.

P. Wadler. 1989. Theorems for Free! In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture, FPCA 1989, London, UK, September 11-13, 1989*, pp. 347–359. DOI: 10.1145/99370.99404.

M. Wenzel. 2002. *Isabelle/Isar — A Versatile Environment for Human-Readable Formal Proof Documents*. PhD thesis, Institut für Informatik, Technische Universität München. https://mediatum.ub.tum.de/?id=601724.

J. Williams. 1964. Algorithm 232 — Heapsort. *Communications of the ACM*, 7(6): 347–348. https://doi.org/10.1145/512274.512284.

S. Wimmer, S. Hu, and T. Nipkow. 2018a. Monadification, memoization and dynamic programming. *Archive of Formal Proofs*. https://isa-afp.org/entries/Monad_Memo_DP.html, Formal proof development.

S. Wimmer, S. Hu, and T. Nipkow. 2018b. Verified memoization and dynamic programming. In J. Avigad and A. Mahboubi, eds., *Interactive Theorem Proving (ITP 2018)*, volume 10895 of *Lecture Notes in Computer Science*, pp. 579–596. Springer. https://doi.org/10.1007/978-3-319-94821-8_34.

N. Wirth. 1971. Program Development by Stepwise Refinement. *Commun. ACM*, 14(4): 221–227. DOI: 10.1145/362575.362577.

D. S. Wise. 1985. Representing matrices as quadtrees for parallel processors. *Inf. Process. Lett.*, 20(4): 195–199. https://doi.org/10.1016/0020-0190(85)90049-3.

D. S. Wise. 1986. Parallel decomposition of matrix inversion using quadtrees. In *International Conference on Parallel Processing, ICPP'86*, pp. 92–99. IEEE Computer Society Press.

D. S. Wise. 1987. Matrix algebra and applicative programming. In G. Kahn, ed., *Functional Programming Languages and Computer Architecture*, volume 274 of *LNCS*, pp. 134–153. Springer. https://doi.org/10.1007/3-540-18317-5_9.

F. F. Yao. 1980. Efficient dynamic programming using quadrangle inequalities. In *Symposium on Theory of Computing, STOC*, pp. 429–435. ACM. https://doi.org/10.1145/800141.804691.

B. Zhan. 2018. Efficient verification of imperative programs using auto2. In D. Beyer and M. Huisman, eds., *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2018*, volume 10805 of *LNCS*, pp. 23–40. Springer. https://doi.org/10.1007/978-3-319-89960-2_2.

Authors

Mohammad Abdulaziz
Department of Informatics
King's College London

Jasmin Blanchette
Institut für Informatik
Ludwig-Maximilians-Universität München

Manuel Eberl
Department of Computer Science
University of Innsbruck

Alejandro Gómez-Londoño[1]
Department of Computer Science and Engineering
Chalmers University of Technology

Peter Lammich
Electrical Engineering, Mathematics and Computer Science
University of Twente

Tobias Nipkow
Department of Computer Science
Technical University of Munich

Lawrence C. Paulson
Computer Laboratory
University of Cambridge

Christian Sternagel[1]
Department of Computer Science
University of Innsbruck

Simon Wimmer[1]
Department of Computer Science
Technical University of Munich

Bohua Zhan[1]
Institute of Software
Chinese Academy of Sciences

[1] Research conducted while at the given address

Index