Calculating Datastructures

Ralf Hinze[1] and Wouter Swierstra[2]

[1] TU Kaiserslautern ralf-hinze@cs.uni-kl.de
[2] Utrecht University w.s.swierstra@uu.nl

Abstract. Where do datastructures come from? This paper explores how to systematically derive implementations of *one-sided flexible arrays* from a simple reference implementation. Using the dependently typed programming language Agda, each calculation constructs an isomorphic—yet more efficient—datastructure using only a handful of laws relating types and arithmetic. Although these calculations do not generally produce novel datastructures they do give insight into how certain datastructures arise and how different implementations are related.

1 Introduction

There is a rich field of *program calculation*, deriving a program systematically from its specification. In this paper, we explore a slightly different problem: showing how efficient *datastructures* can be derived from an inefficient reference implementation. In particular, we consider how to derive implementations of *one-sided flexible arrays*, that offer efficient indexing without being limited to store only a fixed number of elements. Although we do not claim to invent new datastructures by means of our calculations, we can demystify the definitions of familiar datastructures, providing a constructive rationalization identifying the key design choices that are made.

In contrast to program calculation, relating a program and its specification, the calculation of datastructures requires relating two different *types*. As it turns out, we show how to calculate efficient implementations that are *isomorphic* to our reference implementation. These calculations rely exclusively on familiar laws of types and arithmetic. Indeed, we have formalised these calculations in the dependently typed programming language and proof assistant Agda. While we present our derivations in quite some detail, we occassionally will refer to the accompanying source code for a more complete account; while not provable in the current type theory underlying Agda, we will occassionally assume the axiom of functional extensionality.

After defining the interface of flexible arrays (Section 2), we will define the Peano natural numbers (Section 3), leading to the first functional reference implementation of flexible arrays (Section 4). Starting from this reference implementation, we compute an isomorphic, yet inefficient, datastructure (Section 5). By shifting to a more efficient (binary) number representation (Section 6), we can define a similar reference implementation (Section 7). Using this second reference implementation, we once again compute an isomorphic datastructure (Section 8)—but in this case several alternative choices exist (Sections 9 & 10).

2 One-sided Flexible Arrays

Consider the following interface for one-sided flexible arrays:

$$
\begin{array}{ll}
\mathbb{N} & : \mathit{Set} \\
\mathit{Array} & : \mathbb{N} \to \mathit{Set} \to \mathit{Set} \\
\mathit{lookup} & : \mathit{Array}\, n\, \mathit{elem} \to (\{\, i : \mathbb{N} \mid i < n \,\} \to \mathit{elem}) \\
\mathit{tabulate} & : (\{\, i : \mathbb{N} \mid i < n \,\} \to \mathit{elem}) \to \mathit{Array}\, n\, \mathit{elem} \\
\mathit{nil} & : \mathit{Array}\, 0\, \mathit{elem} \\
\mathit{cons} & : \mathit{elem} \to \mathit{Array}\, n\, \mathit{elem} \to \mathit{Array}\, (1 + n)\, \mathit{elem} \\
\mathit{head} & : \mathit{Array}\, (1 + n)\, \mathit{elem} \to \mathit{elem} \\
\mathit{tail} & : \mathit{Array}\, (1 + n)\, \mathit{elem} \to \mathit{Array}\, n\, \mathit{elem}
\end{array}
$$

An array of type $\mathit{Array}\, n\, \mathit{elem}$ stores n elements of type elem, for some natural number n. For the moment, we leave the type of natural numbers *abstract*. In what follows, we explore different implementations of arrays by varying the implementation of the natural number type.

We require flexible arrays to be isomorphic to functions from some finite set of indices to the *elem* type. The *lookup* function witnesses one direction of the isomorphism, *tabulate* the other.

$$\mathit{lookup}\, (\mathit{tabulate}\, f) \equiv f \tag{1a}$$

$$\mathit{tabulate}\, (\mathit{lookup}\, a) \equiv a \tag{1b}$$

In what follows, we refer to functions with a finite domain, that is functions of the form $\{\, i : \mathbb{N} \mid i < n \,\} \to \mathit{elem}$, as *finite maps*.

In contrast to traditional fixed-size arrays, one-sided flexible arrays can be extended at the front using the *cons* operation. Non-empty arrays can be shrunk with *tail*, discarding the first element. The following properties specify the interplay of indexing and the other operations that modify the size of the array.

$$\mathit{lookup}\, (\mathit{cons}\, x\, xs)\, 0 \equiv x \tag{2a}$$

$$\mathit{lookup}\, (\mathit{cons}\, x\, xs)\, (1 + i) \equiv \mathit{lookup}\, xs\, i \tag{2b}$$

$$\mathit{head}\, xs \equiv \mathit{lookup}\, xs\, 0 \tag{2c}$$

$$\mathit{lookup}\, (\mathit{tail}\, xs)\, i \equiv \mathit{lookup}\, xs\, (1 + i) \tag{2d}$$

To define any implementation of this interface, we first need to settle on the implementation of the natural number type. The most obvious choice is, of course, Peano's representation.

3 Peano Numerals

To calculate an implementation of flexible arrays we proceed in two steps. First, we fix an indexing scheme by defining a type of natural numbers below some fixed, upper bound. Such an indexing scheme fixes the domain of our finite maps,

$\{i \mid i < n\}$. Next, we calculate a more efficient representation of finite maps, yielding a datastructure rather than a function. This section details the ideas underlying the first step using the simplest representation of natural numbers; in Section 5, we explore the second step.

3.1 Number Type

The datatype of Peano numerals describes the set of natural numbers as the least set containing *zero* that is closed under a *succ*essor operation.

```
data Peano : Set where
  zero : Peano
  succ : Peano → Peano
```

We use variable names such as k, m, and n to range over Peano numerals and use the Arabic numerals to denote *Peano* constants, writing 3 rather than $succ\,(succ\,(succ\,zero))$.

The operations doubling and incrementing natural numbers, needed in Section 6.1, illustrate how to define functions (by induction) in Agda.[3]

```
_+1 _+2 : Peano → Peano
n +1 = succ n
n +2 = succ (succ n)
_·2 : Peano → Peano
zero ·2 = zero
succ n ·2 = succ (succ (n ·2))
```

The underscores indicate that all three functions are written postfix.

3.2 Index Type

Having fixed the number type, we move on to define the type of valid indices in an *Array* of size n. Here we have several alternatives, each with its own advantages and disadvantages. The most obvious transcription of $\{i \mid i < n\}$ uses a *dependent pair* or Σ type to combine a natural number and a proof that it is within bounds:

```
Index : Peano → Set
Index n = Σ[ i ∈ Peano ] i < n
```

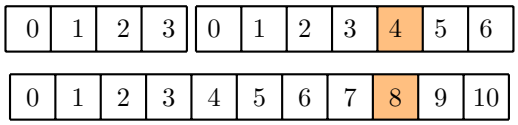
[3] The single most important rule when reading Agda code is that *only* a space separates lexemes. For example, +1 is a single lexeme denoting the successor function, whereas $n + 1$ is a sequence of three lexemes. In general, an Agda identifier consists of an arbitrary sequence of non-whitespace Unicode characters. There are only a few exceptions to this rule: for example, parentheses, (and), and curly braces, { and }, must not form part of a name—for obvious reasons.

Here $<$ denotes the *strict* ordering on the naturals. While the definition is fairly straightforward, it is somewhat cumbersome to use in practice as any computation on indices involves manipulations of proofs. Before discussing alternative definitions, let us first explore some properties of the *Index* type.

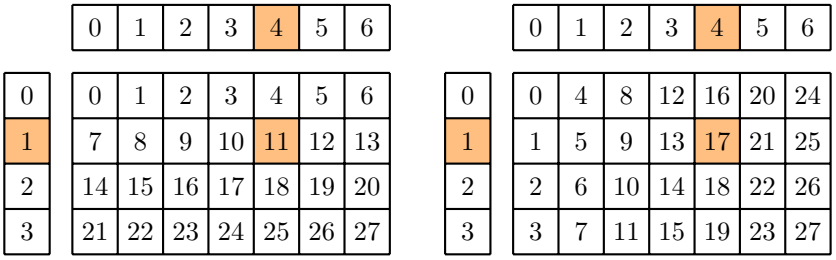
$$\begin{array}{lll} \mathit{Index}\text{-}0 & : \mathit{Index}\,(0) & \cong \bot \\ \mathit{Index}\text{-}1 & : \mathit{Index}\,(1) & \cong \top \\ \mathit{Index}\text{-}+ & : \mathit{Index}\,(m+n) & \cong (\mathit{Index}\,m \uplus \mathit{Index}\,n) \\ \mathit{Index}\text{-}* & : \mathit{Index}\,(m*n) & \cong (\mathit{Index}\,m \times \mathit{Index}\,n) \\ \mathit{Index}\text{-}\uparrow & : \mathit{Index}\,(n \uparrow m) & \cong (\mathit{Index}\,m \to \mathit{Index}\,n) \end{array}$$

The formulas link arithmetic on numbers to operations on types, with the number 0 corresponding to the empty set (written $\bot$ in Agda), 1 to a singleton set (written $\top$), addition to disjoint union (written $\uplus$), multiplication to cartesian product (written $\times$), and, finally, exponentiation to the function space. We refer to these laws as *index transformations*.

For example, the sum rule *Index*-$+$ is witnessed by a mapping between indices for a pair of arrays and indices for a single array, as suggested below.



More instructively, the product rule, *Index*-$*$, is witnessed by a mapping between indices for a two-dimensional array and indices for a one-dimensional array.



In general, there is not a single *canonical* witness for an index transformation. The diagram on the left above exemplifies what is known as row-major order, but there is also column-major order, shown on the right. For now, we choose to ignore these specifics. However, when we start calculating datastructures the *choice* of isomorphism becomes tremendously important, a point we return to in Section 6.2.

Remark 1 (Categorical background). To provide some background, the type function *Index* is the object part of a functor from the preorder of natural numbers to the category of finite sets and total functions. (This is why the type is also known as *Fin* or *FinSet*.) The action of the functor on arrows embeds $\mathit{Index}\,m$ into $\mathit{Index}\,n$, provided $m \preccurlyeq n$. In fact, the isomorphisms demonstrate that *Index*

is simultaneously a *strong monoidal* functor of type $(\mathbb{N}, 0, +) \to (\mathit{Set}, \bot, \uplus)$ and a *strong monoidal* functor of type $(\mathbb{N}, 1, \cdot) \to (\mathit{Set}, \top, \times)$. □

Returning to the issue of defining the *Index* type in Agda, we can use the isomorphisms above to determine the index set by pattern matching on the natural number n.

$$\begin{aligned}
&\mathit{Index} : \mathit{Peano} \to \mathit{Set}\\
&\mathit{Index}\,(\mathit{zero}) \;\; = \bot\\
&\mathit{Index}\,(\mathit{succ}\,n) \;= \top \uplus \mathit{Index}\,n
\end{aligned}$$

The zero rule *Index*-0 determines that the $\mathit{Index}\,(0)$ type is uninhabited; whereas the rules for one and addition, *Index*-1 and *Index*-+, determine that $\mathit{Index}\,(\mathit{succ}\,n)$ contains one more element than $\mathit{Index}\,n$.

For reasons of readability, we turn the definition of *Index* into idiomatic Agda, replacing the type function by an inductively defined indexed datatype.

$$\begin{aligned}
&\textbf{\textit{data}}\ \mathit{Index} : \mathit{Peano} \to \mathit{Set}\ \textbf{\textit{where}}\\
&\quad \mathit{izero} : \qquad\quad \mathit{Index}\,(\mathit{succ}\,n)\\
&\quad \mathit{isucc} : \mathit{Index}\,n \to \mathit{Index}\,(\mathit{succ}\,n)
\end{aligned}$$

There are no constructors for $\mathit{Index}\,\mathit{zero}$, corresponding to the first equation of *Index*, and two constructors for $\mathit{Index}\,(\mathit{succ}\,n)$, corresponding to *Index*'s second equation. The constructor names are almost identical to those of *Peano*. This is intentional: we want the constructors of *Index* to look *and* behave like their namesakes. The only difference is that the former carry vital type information about upper bounds. Reassuringly, all three definitions of index sets are equivalent. The straightforward, but rather technical proofs can be found in the accompanying material.

To illustrate working with indices, let us implement some index transformations that are needed later in Section 6.2.

$$\begin{aligned}
&_{\cdot}2{+}0\;_{\cdot}2{+}1 : \mathit{Index}\,n \to \mathit{Index}\,(n \cdot 2)\\
&(\mathit{izero}) \;\; \cdot 2{+}0 \;=\; \mathit{izero}\\
&(\mathit{isucc}\,i)\cdot 2{+}0 \;=\; \mathit{isucc}\,(\mathit{isucc}\,(i \cdot 2{+}0))\\
&(\mathit{izero}) \;\; \cdot 2{+}1 \;=\; \mathit{isucc}\,\mathit{izero}\\
&(\mathit{isucc}\,i)\cdot 2{+}1 \;=\; \mathit{isucc}\,(\mathit{isucc}\,(i \cdot 2{+}1))
\end{aligned}$$

The second operation combines doubling and increment. The "obvious" definition, $i \cdot 2{+}1 \;=\; \mathit{isucc}\,(i \cdot 2{+}0)$ does not work, as the expression on the right-hand side has type $\mathit{Index}\,(n \cdot 2 + 1)$ and not $\mathit{Index}\,(n \cdot 2)$, as required. On plain naturals we can separate doubling and increment; here we need to combine the operations to be able to establish precise upper bounds. We cannot expect Agda to automatically replicate the hand-written proof:

$$i \prec n \quad\Longleftrightarrow\quad i + 1 \preccurlyeq n \quad\Longleftrightarrow\quad (i+1)\cdot 2 \preccurlyeq n \cdot 2 \quad\Longleftrightarrow\quad i \cdot 2 + 1 \prec n \cdot 2\ .$$

In general, since *Index* combines data and proof, index transformations require more work than their vanilla counterparts on naturals.

Now that we have a precise understanding of the domain of our finite maps, we can start calculating an implementation of the interface specified in Section 2.

4 Functions as Datastructures

The simplest implementation of the *Array* type identifies arrays and finite maps:

$$\begin{aligned}&\mathit{Array} \;:\; \mathit{Peano} \to \mathit{Set} \to \mathit{Set}\\ &\mathit{Array}\; n\; \mathit{elem} \;=\; \mathit{Index}\; n \to \mathit{elem}\end{aligned}$$

In this particular case, *lookup* and *tabulate* are manifest identities, rather than isomorphisms.

$$\begin{aligned}&\mathit{lookup} \;:\; \mathit{Array}\; n\; \mathit{elem} \to (\mathit{Index}\; n \to \mathit{elem})\\ &\mathit{lookup}\; a \;=\; a\\ &\mathit{tabulate} \;:\; (\mathit{Index}\; n \to \mathit{elem}) \to \mathit{Array}\; n\; \mathit{elem}\\ &\mathit{tabulate}\; f \;=\; f\end{aligned}$$

To complete this "implementation", however, we still need to define the remaining operations: *nil*, *cons*, *head*, and *tail*. The empty array *nil* is the unique function from the empty set, defined below using an absurd pattern, written (). For the other functions, the specification serves as the implementation, for example, (2a) and (2b) form the definition of *cons*, given that *lookup* is implemented as the identity function.

$$\begin{aligned}&\mathit{nil} \;:\; \mathit{Array}\; \mathit{zero}\; \mathit{elem}\\ &\mathit{nil}\; ()\\ &\mathit{cons} \;:\; \mathit{elem} \to \mathit{Array}\; n\; \mathit{elem} \to \mathit{Array}\; (\mathit{succ}\; n)\; \mathit{elem}\\ &\mathit{cons}\; x\; xs\; (\mathit{izero}) \;\;\;=\; x\\ &\mathit{cons}\; x\; xs\; (\mathit{isucc}\; i) \;=\; xs\; i\\ &\mathit{head} \;:\; \mathit{Array}\; (\mathit{succ}\; n)\; \mathit{elem} \;\to\; \mathit{elem}\\ &\mathit{head}\; xs \;=\; xs\; \mathit{izero}\\ &\mathit{tail} \;:\; \mathit{Array}\; (\mathit{succ}\; n)\; \mathit{elem} \;\to\; \mathit{Array}\; n\; \mathit{elem}\\ &\mathit{tail}\; xs\; i \;=\; xs\; (\mathit{isucc}\; i)\end{aligned}$$

The proofs that the "implementation" satisfies the specification are consequently trivial: all the specified equivalences hold by definition.

While the "implementation" is exceptionally simple, it is also exceptionally slow: the running time of *lookup xs i* is not only determined by the index *i* but also by the number of operations used to build the array *xs*. For example, even though *tail* (*cons x xs*) is extensionally equal to *xs*, each lookup takes two additional steps as the index is first incremented by *tail* only to be subsequently decremented by *cons*. In other words, the run-time behaviour of *lookup* is sensitive to *how* the array has been constructed! To avoid this problem, we turn functions into datastructures, using this "implementation" above as our starting point.

5 Lists also Known as Vectors

Do you remember the *laws of exponents* from secondary school?

$$X^0 = 1 \qquad X^1 = X \qquad X^{A+B} = X^A \cdot X^B \qquad X^{A \cdot B} = \left(X^B\right)^A$$

Quite amazingly, these equalities can be re-interpreted as isomorphisms between types, where B^A is the type of functions from A to B.

$$\begin{array}{ll}
\mathit{law\text{-}of\text{-}exponents\text{-}}\bot & : (\bot \to X) \qquad \cong \top \\
\mathit{law\text{-}of\text{-}exponents\text{-}}\top & : (\top \to X) \qquad \cong X \\
\mathit{law\text{-}of\text{-}exponents\text{-}}\uplus & : (A \uplus B \to X) \cong ((A \to X) \times (B \to X)) \\
\mathit{law\text{-}of\text{-}exponents\text{-}}\times & : (A \times B \to X) \cong (A \to (B \to X))
\end{array}$$

If we apply these isomorphisms from left to right, perhaps repeatedly, we can systematically eliminate function spaces. This process might be called *defunctionalization* or *trieification*, except that the former term is already taken.

Some background is perhaps not amiss. A trie is also known as a digital search tree. In a conventional search tree, the search path is determined on the fly by comparing a given key against the sign posts stored in the inner nodes. By contrast, in a digital search tree, the key *is* the search path. This idea is, of course, not limited to searching. The point of this paper is that it applies equally well to indexing: the index or position of an element within an array *is* a path into the datastructure that represents the array.

Remark 2 (Categorical background). A lot more can be said about trieification: *Tab* with $\mathit{Tab}\,A = A \to X$ is a *contravariant* functor, part of an adjoint situation, sending left adjoints (initial objects $\bot$, coproducts $\uplus$, initial algebras) to right adjoints (terminal objects $\top$, products $\times$, final coalgebras) [2, 13]. □

That's enough words for the moment, *calculemus*! To trieify our type of finite maps,

$$\mathit{trieify} : \forall\, \mathit{elem} \to \forall\, n \to (\mathit{Index}\, n \to \mathit{elem}) \cong \mathit{Array}\, n\, \mathit{elem}$$

we proceed by induction on the size of the array n. For the derivation, we use Wim Feijen's proof format, which features explicit justifications for the calculational steps, written between angle brackets. For example, the first rewrite below is justified by an isomorphism between function spaces: $\mathit{dom} \cong\!\to\!\cong \mathit{cod}$ applies the isomorphism *dom* to the domain of its function argument and *cod* to its codomain. Note that in contrast to traditional pencil-and-paper proofs, the justification is an Agda term, and the Agda type-checker verifies that this term serves indeed as appropriate evidence for the step.

$$\begin{array}{l}
\mathit{trieify}\ \mathit{elem}\ \mathit{zero} = \\
\quad \textbf{\textit{proof}} \\
\qquad (\mathit{Index}\ \mathit{zero} \to \mathit{elem}) \\
\quad \cong\langle\ \mathit{Index\text{-}zero} \cong\!\to\!\cong \cong\text{-}\mathit{reflexive}\ \rangle \\
\qquad (\bot \to \mathit{elem}) \\
\quad \cong\langle\ \mathit{law\text{-}of\text{-}exponents\text{-}}\bot\ \rangle \\
\qquad \top \\
\quad \cong\langle\ \mathit{use\text{-}as\text{-}definition\text{-}of}\ \mathit{Array\text{-}zero}\ \rangle \\
\qquad \mathit{Array}\ \mathit{zero}\ \mathit{elem} \\
\quad \blacksquare
\end{array}$$

The calculation suggests a defining equation: $\mathit{Array}\ \mathit{zero}\ \mathit{elem} = \top$, which expresses that there is exactly one array of size 0, namely the empty array. For non empty arrays, the calculation is almost just as straightforward.

$$
\begin{array}{l}
\mathit{trieify}\ \mathit{elem}\ (\mathit{succ}\ n) = \\
\quad \textbf{\textit{proof}} \\
\qquad (\mathit{Index}\ (\mathit{succ}\ n) \to \mathit{elem}) \\
\quad \cong \langle\ \mathit{Index\text{-}succ} \cong \to \cong \cong \text{-}\mathit{reflexive}\ \rangle \\
\qquad (\top \uplus \mathit{Index}\ n \to \mathit{elem}) \\
\quad \cong \langle\ \mathit{law\text{-}of\text{-}exponents\text{-}}\uplus\ \rangle \\
\qquad (\top \to \mathit{elem}) \times (\mathit{Index}\ n \to \mathit{elem}) \\
\quad \cong \langle\ \mathit{law\text{-}of\text{-}exponents\text{-}}\top \cong \times \cong \mathit{trieify}\ \mathit{elem}\ n\ \rangle \\
\qquad \mathit{elem} \times \mathit{Array}\ n\ \mathit{elem} \\
\quad \cong \langle\ \mathit{use\text{-}as\text{-}definition\text{-}of}\ \mathit{Array\text{-}succ}\ \rangle \\
\qquad \mathit{Array}\ (\mathit{succ}\ n)\ \mathit{elem} \\
\quad \blacksquare
\end{array}
$$

The final isomorphism expresses that an array of size $1 + n$ consists of an element followed by an array of size n. If we name the constructors appropriately, we obtain the familiar datatype of lists, indexed by length. This indexed type is also known as *Vector*.

$$
\begin{array}{l}
\textbf{\textit{variable}}\ \mathit{elem} : \mathit{Set} \\
\textbf{\textit{data}}\ \mathit{Array} : \mathit{Peano} \to \mathit{Set} \to \mathit{Set}\ \textbf{\textit{where}} \\
\quad \mathit{nil} \quad : \mathit{Array}\ \mathit{zero}\ \mathit{elem} \\
\quad \mathit{cons} : \mathit{elem} \to \mathit{Array}\ n\ \mathit{elem} \to \mathit{Array}\ (\mathit{succ}\ n)\ \mathit{elem}
\end{array}
$$

Observe that the constructors of the interface, *nil* and *cons*, are now implemented by the constructors of the datatype.

If we *extract* the two components of the *trieify* isomorphism, we obtain the following definitions of *lookup* and *tabulate*.[4]

$$
\begin{array}{l}
\mathit{lookup} : \mathit{Array}\ n\ \mathit{elem} \to (\mathit{Index}\ n \to \mathit{elem}) \\
\mathit{lookup}\ (\mathit{cons}\ x\ \mathit{xs})\ (\mathit{izero}) \quad = x \\
\mathit{lookup}\ (\mathit{cons}\ x\ \mathit{xs})\ (\mathit{isucc}\ i) = \mathit{lookup}\ \mathit{xs}\ i \\
\\
\mathit{tabulate} : (\mathit{Index}\ n \to \mathit{elem}) \to \mathit{Array}\ n\ \mathit{elem} \\
\mathit{tabulate}\ \{\mathit{zero}\} \quad \mathit{fm} = \mathit{nil} \\
\mathit{tabulate}\ \{\mathit{succ}\ n\}\ \mathit{fm} = \mathit{cons}\ (\mathit{fm}\ \mathit{izero})\ (\mathit{tabulate}\ (\lambda\, i \to \mathit{fm}\ (\mathit{isucc}\ i)))
\end{array}
$$

Like *trieify*, both *lookup* and *tabulate* are defined by induction on the size. In the case of *lookup*, the size information remains implicit as Agda is able to recreate it from the explicit argument, namely, the argument list. For *tabulate*, no such information is available. Hence, we need to match on the implicit argument explicitly in its definition.

[4] Unfortunately, Agda's extraction process is only semi-automatic, so we do not trust the resulting code. The proofs that *lookup* and *tabulate* are inverses are, however, entirely straightforward and can be found in the accompanying material.

The equations for *head* and *tail* are immediate consequences of the specification.

$$\begin{aligned}
&\mathit{head} : \mathit{Array}\,(\mathit{succ}\,n)\;\mathit{elem} \to \mathit{elem}\\
&\mathit{head}\,(\mathit{cons}\,x\,\mathit{xs}) = x\\
&\mathit{tail} : \mathit{Array}\,(\mathit{succ}\,n)\;\mathit{elem} \to \mathit{Array}\,n\;\mathit{elem}\\
&\mathit{tail}\,(\mathit{cons}\,x\,\mathit{xs}) = \mathit{xs}
\end{aligned}$$

Thanks to our reference implementation, see Section 4, the proof of correctness is a breeze: we simply show that the "concrete" operations on vectors are equivalent to their "specification" on finite maps. Defining a shortcut for the *lookup* function, $[\![\_]\!]$, the proof obligations read as follows.

$$[\![\,\mathit{nil}\,]\!] \equiv \mathit{FM.nil} \tag{3a}$$
$$[\![\,\mathit{cons}\,x\,\mathit{xs}\,]\!] \equiv \mathit{FM.cons}\,x\,[\![\,\mathit{xs}\,]\!] \tag{3b}$$
$$\mathit{head}\,\mathit{xs} \equiv \mathit{FM.head}\,[\![\,\mathit{xs}\,]\!] \tag{3c}$$
$$[\![\,\mathit{tail}\,\mathit{xs}\,]\!] \equiv \mathit{FM.tail}\,[\![\,\mathit{xs}\,]\!] \tag{3d}$$
$$\mathit{lookup}\,\mathit{xs} \equiv \mathit{FM.lookup}\,[\![\,\mathit{xs}\,]\!] \tag{3e}$$
$$[\![\,\mathit{tabulate}\,f\,]\!] \equiv \mathit{FM.tabulate}\,f \tag{3f}$$

Here we have prefixed the operations of the reference implementation of Section 4 by *FM*, short for 'finite map'. Recalling that *FM.lookup* and *FM.tabulate* are both the identity function, Property (3e) means that *lookup* is indeed extensionally equal to the implementation using finite maps. Conversely, as (3f) shows, the *tabulate* is the right-inverse of *lookup*, which is unique.

The vector implementation of arrays does not suffer from history-sensitivity, $\mathit{tail}\,(\mathit{cons}\,x\,\mathit{xs})$ is now definitionally equal to *xs*, but thanks to the ivory tower number type, it is still too slow to be useful in practice. The cure is pretty obvious: we replace unary numbers by binary numbers—albeit with a twist.

6 Leibniz Numerals

Instead of working with Peano naturals, we could choose a different implementation of the natural number type. In this section, we will explore one possible implementation, *Leibniz numbers*, or binary numbers that have a unique representation of every Peano natural number.

6.1 Number Type

A Leibniz numeral is given by a sequence of digits with the most significant digit on the left. A digit is either `1` or `2`.

$$\begin{aligned}
&\textbf{\textit{data}}\ \mathit{Leibniz} : \mathit{Set}\ \textbf{\textit{where}}\\
&\quad \mathtt{0}b : \mathit{Leibniz}\\
&\quad _\mathtt{1} : \mathit{Leibniz} \to \mathit{Leibniz}
\end{aligned}$$

```
  _2 : Leibniz → Leibniz
Eau-de-Cologne = 0b 1 1 2 1 1 2 2 1 2 1 1 1
```

Agda's postfix syntax allows us to mimic standard notation for binary numbers: the expression `0b 1 1 2 1` should be read as `(((0b 1) 1) 2) 1`.

To assign a meaning to a Leibniz numeral, we map it to a Peano numeral.

```
𝒩⟦_⟧ : Leibniz → Peano
𝒩⟦ 0b ⟧ = 0
𝒩⟦ n 1 ⟧ = 𝒩⟦ n ⟧ ·2 +1
𝒩⟦ n 2 ⟧ = 𝒩⟦ n ⟧ ·2 +2
assert : 𝒩⟦ Eau-de-Cologne ⟧ ≡ 4711
```

For example, $\mathcal{N}[\![\ \texttt{0b 1 1 2 1}\ ]\!]$ normalizes to 17. The meaning function makes it crystal clear that we implement a base-two positional number system, except that the digits `1` and `2` are used, rather than `0` and `1`.

Thanks to this twist we avoid the problem of leading zeros: every natural number enjoys a *unique* representation; the Leibniz number system is *non-redundant*. Moreover, the meaning function establishes a one-to-one correspondence between the two number systems: $\mathit{Leibniz} \cong \mathit{Peano}$. Speaking of number conversion, the other direction of the isomorphism can be easily implemented using the "pseudo-constructors" *zero* and *succ*.

```
zero : Leibniz
zero = 0b
succ : Leibniz → Leibniz
succ (0b)  = 0b 1
succ (n 1) = n 2
succ (n 2) = (succ n) 1   -- carry
```

The binary increment exhibits the typical recursion pattern: the least significant digit is incremented, unless it is maximal, in which case a carry is propagated to the left. Using the meaning function it is straightforward to show that the implementation is correct.

```
zero-correct : 𝒩⟦ Leibniz.zero   ⟧ ≡ Peano.zero
succ-correct : 𝒩⟦ Leibniz.succ n ⟧ ≡ Peano.succ 𝒩⟦ n ⟧
```

The prefix "pseudo" indicates that the operations *zero* and *succ* are not full-fledged constructors: we cannot use them in patterns on the left-hand side of definitions. To compensate for this, we additionally offer a *Peano view* [18, 27].

```
data Peano-View : Leibniz → Set where
  as-zero : Peano-View zero
  as-succ : (i : Leibniz) → Peano-View (succ i)
```

The *view* function itself illustrates the use of view patterns.

$$
\begin{array}{l}
\mathit{view} : (n : \mathit{Leibniz}) \to \mathit{Peano\text{-}View}\ n \\
\mathit{view}\ (0\, b) \quad\ = \mathit{as\text{-}zero} \\
\mathit{view}\ (n\, 1)\ \textbf{\textit{with}}\ \mathit{view}\ n \\
\ldots \mid \mathit{as\text{-}zero} \quad = \mathit{as\text{-}succ}\ 0\, b \\
\ldots \mid \mathit{as\text{-}succ}\ m\ = \mathit{as\text{-}succ}\ (m\, 2) \quad \text{-- borrow} \\
\mathit{view}\ (n\, 2) \quad\ = \mathit{as\text{-}succ}\ (n\, 1)
\end{array}
$$

In a sense, *view* combines two functions: the test for zero and the predecessor function, again following the typical recursion pattern: the least significant digit is decremented, unless it is minimal, in which case we borrow one from the left.

The semantics of such a view is defined by a mapping into the Peano numerals. The correctness criterion asserts that *view* does not change the value of its argument.

$$
\begin{array}{l}
\mathscr{V}[\![\_]\!] : \mathit{Peano\text{-}View}\ n \to \mathit{Peano} \\
\mathscr{V}[\![\ \mathit{as\text{-}zero}\ \ ]\!] = 0 \\
\mathscr{V}[\![\ \mathit{as\text{-}succ}\ n\ ]\!] = \mathscr{N}[\![\ n\ ]\!] + 1 \\
\mathit{view\text{-}correct} : \mathscr{V}[\![\ \mathit{view}\ n\ ]\!] \equiv \mathscr{N}[\![\ n\ ]\!]
\end{array}
$$

Remark 3 (Agda). You may wonder why the type *Peano-View* is indexed by a Leibniz numeral. Why not simply define:

$$
\begin{array}{l}
\textbf{\textit{data}}\ \mathit{Peano\text{-}View} : \mathit{Set}\ \textbf{\textit{where}} \quad \text{-- too simple-minded} \\
\quad \mathit{as\text{-}zero} : \mathit{Peano\text{-}View} \\
\quad \mathit{as\text{-}succ} : \mathit{Leibniz} \to \mathit{Peano\text{-}View}
\end{array}
$$

In contrast to the simple, unindexed datatype above, our indexed view type keeps track of the to-be-viewed value, which turns out to be vital for correctness proofs: if *view n* yields *as-succ m* then we know that *n* definitionally equals *succ m*. The constructors of the unindexed datatype do not maintain this important piece of information, so the subsequent proofs do not go through. □

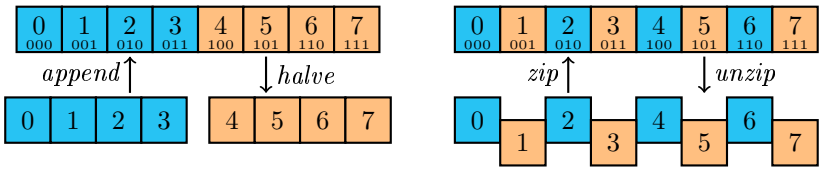
As an intermediate summary, Leibniz numerals serve as a drop-in replacement for Peano numerals: the pseudo-constructors replace *zero* and *succ* on the right-hand side of equations; the view allows us to additionally replace them in patterns on the left-hand side.

6.2 Index Type

Of course, we would like to use binary numbers for indices, as well. Therefore we need to adapt the type of positions that specifies the domain of our finite maps. The following derivation is based on the *index transformations* but, as we have noted in Section 3, there are, in general, several options for the witnesses of these transformations. In other words, we have to make some design decisions! In particular, since we use a binary, *positional* number system we need to inject life into the *doubling* isomorphism:

$$\mathit{Index\text{-}2{\cdot}n{\cong}Index\text{-}n{\uplus}Index\text{-}n} \;:\; \mathit{Index}\,(n \cdot 2) \cong \mathit{Index}\,n \uplus \mathit{Index}\,n$$

There are two canonical choices, one based on appending and a second one based on zipping or interleaving.



If the size is an exact power of two, halving separates the indices based on the most significant bit, whereas unzipping considers the least significant bit. (As an aside, do not be confused by the names: *append*, *zip* etc are index transformations, not operations on arrays.) Both choices are viable, however, we choose to initially focus on the first alternative and return to the second in Section 12, but only briefly.

Zipping maps elements of the first summand to even indices and elements of the second to odd indices.

$$\begin{aligned}
&\mathit{zip} \;:\; \mathit{Index}\,n \uplus \mathit{Index}\,n \;\to\; \mathit{Index}\,(n \cdot 2)\\
&\mathit{zip}\,(\mathit{inj}_1\;i) \;=\; i \cdot 2{+}0 \quad \text{-- even}\\
&\mathit{zip}\,(\mathit{inj}_2\;i) \;=\; i \cdot 2{+}1 \quad \text{-- odd}
\end{aligned}$$

Its inverse amounts to division by 2 with the remainder specifying the summand of the disjoint union.

$$\begin{aligned}
&\mathit{unzip} \;:\; \mathit{Index}\,(n \cdot 2) \;\to\; \mathit{Index}\,n \uplus \mathit{Index}\,n\\
&\mathit{unzip}\,\{\mathit{succ}\,n\}\,(\mathit{izero}) \;=\; \mathit{inj}_1\;\mathit{izero} \quad \text{-- even}\\
&\mathit{unzip}\,\{\mathit{succ}\,n\}\,(\mathit{isucc}\,\mathit{izero}) \;=\; \mathit{inj}_2\;\mathit{izero} \quad \text{-- odd}\\
&\mathit{unzip}\,\{\mathit{succ}\,n\}\,(\mathit{isucc}\,(\mathit{isucc}\,i))\;\textbf{\textit{with}}\;\mathit{unzip}\;i\\
&\ldots \mid \mathit{inj}_1\;j \;=\; \mathit{inj}_1\;(\mathit{isucc}\,j)\\
&\ldots \mid \mathit{inj}_2\;k \;=\; \mathit{inj}_2\;(\mathit{isucc}\,k)
\end{aligned}$$

Given these prerequisites the calculation of

$$\mathit{re\text{-}index} \;:\; \forall\, n \;\to\; \mathit{Peano.Index}\,[\![\, n \,]\!] \cong \mathit{Leibniz.Index}\,n$$

proceeds by induction on the structure of Leibniz numerals. For the base case, there is little to do.

$$\begin{aligned}
&\mathit{re\text{-}index}\,(0b) \;=\\
&\quad \textbf{\textit{proof}}\\
&\qquad \mathit{Peano.Index}\,[\![\, 0b \,]\!]\\
&\quad \cong\langle\; \mathit{Peano.Index\text{-}zero} \;\rangle\\
&\qquad \bot
\end{aligned}$$

$\cong\langle$ *use-as-definition-of Index*-0 $\rangle$
Leibniz.Index 0*b*
■

The calculation for the first inductive case also works without any surprises.

re-index $(n\,1)$ =
proof
Peano.Index $[\![\, n\,1\,]\!]$
$\cong\langle$ *Peano.Index-succ* $\rangle$
$\top \uplus$ *Peano.Index* $([\![\, n\,]\!] \cdot 2)$
$\cong\langle$ $\cong$*-reflexive* $\cong\uplus\cong$ *Index-2·n*$\cong$*Index-n*$\uplus$*Index-n* $\rangle$
$\top \uplus$ *Peano.Index* $[\![\, n\,]\!] \uplus$ *Peano.Index* $[\![\, n\,]\!]$
$\cong\langle$ $\cong$*-reflexive* $\cong\uplus\cong$ (*re-index n* $\cong\uplus\cong$ *re-index n*) $\rangle$
$\top \uplus$ *Leibniz.Index n* $\uplus$ *Leibniz.Index n*
$\cong\langle$ *use-as-definition-of Index*-1 $\rangle$
Leibniz.Index $(n\,1)$
■

We plug in the definition of *Peano.Index*, apply the doubling isomorphism based on zipping, and finally invoke the induction hypothesis. The derivation for the final case follows exactly the same pattern, except that we unfold the definition of *Peano.Index* twice.

re-index $(n\,2)$ =
proof
Peano.Index $[\![\, n\,2\,]\!]$
$\cong\langle$ *Peano.Index-succ* $\rangle$
$\top \uplus$ *Peano.Index* $([\![\, n\,]\!] \cdot 2 + 1)$
$\cong\langle$ $\cong$*-reflexive* $\cong\uplus\cong$ *Peano.Index-succ* $\rangle$
$\top \uplus \top \uplus$ *Peano.Index* $([\![\, n\,]\!] \cdot 2)$
$\cong\langle$ $\cong$*-reflexive* $\cong\uplus\cong$ ($\cong$*-reflexive* $\cong\uplus\cong$ *Index-2·n*$\cong$*Index-n*$\uplus$*Index-n*) $\rangle$
$\top \uplus \top \uplus$ *Peano.Index* $[\![\, n\,]\!] \uplus$ *Peano.Index* $[\![\, n\,]\!]$
$\cong\langle$ $\cong$*-reflexive* $\cong\uplus\cong$ ($\cong$*-reflexive* $\cong\uplus\cong$ (*re-index n* $\cong\uplus\cong$ *re-index n*)) $\rangle$
$\top \uplus \top \uplus$ *Leibniz.Index n* $\uplus$ *Leibniz.Index n*
$\cong\langle$ *use-as-definition-of Index*-2 $\rangle$
Leibniz.Index $(n\,2)$
■

As usual, we introduce names for the summands of the disjoint unions, obtaining the following index type for Leibniz numerals.

data *Index* : *Leibniz* $\to$ *Set* ***where***
$0b_1$: *Index* $(n\,1)$ -- $\top$
$\_1_1$: *Index n* $\to$ *Index* $(n\,1)$ -- $\uplus$ *Index n*
$\_2_1$: *Index n* $\to$ *Index* $(n\,1)$ -- $\uplus$ *Index n*
$0b_2$: *Index* $(n\,2)$ -- $\top$

$$
\begin{array}{lll}
\mathtt{1}b_2 & : & \mathit{Index}\,(n\,\mathtt{2}) \quad \text{-- } \uplus\, \mathit{Index}\,n \\
\_\mathtt{2}_2 & : & \mathit{Index}\,n \to \mathit{Index}\,(n\,\mathtt{2}) \quad \text{-- } \uplus\, \mathit{Index}\,n \\
\_\mathtt{3}_2 & : & \mathit{Index}\,n \to \mathit{Index}\,(n\,\mathtt{2}) \quad \text{-- } \uplus\, \mathit{Index}\,n
\end{array}
$$

A couple of remarks are in order. The index attached to the constructor names indicates the least significant digit of the upper bound. The constructors $\mathtt{0}b_1$ and $\mathtt{0}b_2$ say: "Operationally we are alike, both representing the zeroth index. However, we carry important type information, $\mathtt{0}b_1$ lives below an odd upper bound, whereas $\mathtt{0}b_2$ is below an even bound." The definition of the index set is perhaps not quite what we expected as it amalgamates two different number systems: the by now familiar `1-2` system and a variant that employs `0` and `1` as leading digits and `2` and `3` for non-leading digits.

Remark 4. As an aside, the `2-3` number system is also non-redundant. In general, any binary system that uses the digits $0, \ldots, a$ in the leading position and $a+1$ and $a+2$ for the other positions, enjoys the property that every natural number has a unique representation. □

To make the semantics of these indices precise, we extract the witness for the reverse direction of the *re-index*ing isomorphism (using *iz* and *is* as shorthands for *izero* and *isucc* on Peano indices).

$$
\begin{array}{lcl}
\mathscr{I}[\![\_]\!] & : & \mathit{Index}\,n \to \mathit{Peano.Index}\,[\![\,n\,]\!] \\
\mathscr{I}[\![\,\mathtt{0}b_1\,]\!] & = & \mathit{iz} \\
\mathscr{I}[\![\,i\,\mathtt{1}_1\,]\!] & = & \mathit{is}\,(\mathscr{I}[\![\,i\,]\!]\cdot 2{+}0) \\
\mathscr{I}[\![\,i\,\mathtt{2}_1\,]\!] & = & \mathit{is}\,(\mathscr{I}[\![\,i\,]\!]\cdot 2{+}1) \\
\mathscr{I}[\![\,\mathtt{0}b_2\,]\!] & = & \mathit{iz} \\
\mathscr{I}[\![\,\mathtt{1}b_2\,]\!] & = & \mathit{is}\,\mathit{iz} \\
\mathscr{I}[\![\,i\,\mathtt{2}_2\,]\!] & = & \mathit{is}\,(\mathit{is}\,(\mathscr{I}[\![\,i\,]\!]\cdot 2{+}0)) \\
\mathscr{I}[\![\,i\,\mathtt{3}_2\,]\!] & = & \mathit{is}\,(\mathit{is}\,(\mathscr{I}[\![\,i\,]\!]\cdot 2{+}1))
\end{array}
$$

Just as we saw in Section 3, the expressions $n\cdot 2{+}1$ and $\mathit{is}\,(n\cdot 2{+}0)$ are quite different as they live below different upper bounds: if $j : \mathit{Index}\,a$, then $j\cdot 2{+}1 : \mathit{Index}\,(a\cdot 2)$, whereas $\mathit{is}\,(j\cdot 2{+}0) : \mathit{Index}\,(a\cdot 2+1)$. These types carry just enough information to avoid the infamous "index out of bounds" errors. While the definitions of *Index* and $\mathscr{I}[\![\_]\!]$ may seem quite bulky at first glance, they encode an essential invariant of the indices involved.

The same remark applies to the definition of *izero* and *isucc*.

$$
\begin{array}{lcl}
\mathit{izero} & : & \forall\,\{n\} \;\to\; \mathit{Index}\,(\mathit{succ}\,n) \\
\mathit{izero}\,\{\mathtt{0}b\} & = & \mathtt{0}b_1 \\
\mathit{izero}\,\{n\,\mathtt{1}\} & = & \mathtt{0}b_2 \\
\mathit{izero}\,\{n\,\mathtt{2}\} & = & \mathtt{0}b_1 \\
\mathit{isucc} & : & \mathit{Index}\,n \to \mathit{Index}\,(\mathit{succ}\,n) \\
\mathit{isucc}\,(\mathtt{0}b_1) & = & \mathtt{1}b_2 \\
\mathit{isucc}\,(i\,\mathtt{1}_1) & = & i\,\mathtt{2}_2 \\
\mathit{isucc}\,(i\,\mathtt{2}_1) & = & i\,\mathtt{3}_2
\end{array}
$$

$$
\begin{aligned}
&\mathit{isucc}\,(0b_2) &&= \mathit{izero}\,1_1\\
&\mathit{isucc}\,(1b_2) &&= \mathit{izero}\,2_1\\
&\mathit{isucc}\,(i\,2_2) &&= (\mathit{isucc}\,i)\,1_1\\
&\mathit{isucc}\,(i\,3_2) &&= (\mathit{isucc}\,i)\,2_1
\end{aligned}
$$

The successor function maps an odd number to an even number, and vice versa, correspondingly incrementing the upper bounds. Consequently, arguments and results alternate between the two number systems. This is why $\mathit{isucc}\,(i\,2_2)$ yields $(\mathit{isucc}\,i)\,1_1$, rather than $i\,3_2$. The recursion pattern is interesting: if the argument is below an odd bound, *isucc* returns immediately; a recursive call is only made for indices that live below an even upper bound. We return to this observation in Section 8.

Using the meaning function we can establish the correctness of *izero* and *isucc*.

$$
\begin{aligned}
&\mathit{izero\text{-}correct} : \mathscr{I}[\![\,\mathit{izero}\,\{n\}\,]\!] \equiv \mathit{iz}\\
&\mathit{isucc\text{-}correct} : \mathscr{I}[\![\,\mathit{isucc}\,i\quad\,]\!] \equiv \mathit{is}\,\mathscr{I}[\![\,i\,]\!]
\end{aligned}
$$

Both equations relate expressions of different types, for example, $\mathscr{I}[\![\,\mathit{isucc}\,i\,]\!] : \mathscr{N}[\![\,\mathit{succ}\,n\,]\!]$ whereas $\mathit{is}\,\mathscr{I}[\![\,i\,]\!] : \mathscr{N}[\![\,n\,]\!]+1$. Fortunately, *succ-correct* tells us that both types are *propositionally* equal.[5]

You may have noticed that this section replicates the structure of Section 6.1. It remains to define an appropriate view on Leibniz indices.

$$
\begin{aligned}
&\textbf{\textit{data}}\ \mathit{Index\text{-}View} : \mathit{Index}\,(\mathit{succ}\,n) \to \mathit{Set}\ \textbf{\textit{where}}\\
&\quad \mathit{as\text{-}izero} : \mathit{Index\text{-}View}\,\{n\}\,\mathit{izero}\\
&\quad \mathit{as\text{-}isucc} : (i : \mathit{Index}\,n) \to \mathit{Index\text{-}View}\,(\mathit{isucc}\,i)
\end{aligned}
$$

The type *Index-View* is implicitly parametrized by a Leibniz numeral n and explicitly parametrized by a Leibniz index of type $\mathit{Index}\,(\mathit{succ}\,n)$. The definition of the view function merits careful study.

$$
\begin{aligned}
&\mathit{iview} : \{n : \mathit{Leibniz}\} \to (i : \mathit{Index}\,(\mathit{succ}\,n)) \to \mathit{Index\text{-}View}\,i\\
&\mathit{iview}\,\{0b\}\ (0b_1) &&= \mathit{as\text{-}izero}\\
&\mathit{iview}\,\{n\,1\}\,(0b_2) &&= \mathit{as\text{-}izero}\\
&\mathit{iview}\,\{n\,1\}\,(1b_2) &&= \mathit{as\text{-}isucc}\,0b_1\\
&\mathit{iview}\,\{n\,1\}\,(i\,2_2) &&= \mathit{as\text{-}isucc}\,(i\,1_1)\\
&\mathit{iview}\,\{n\,1\}\,(i\,3_2) &&= \mathit{as\text{-}isucc}\,(i\,2_1)\\
&\mathit{iview}\,\{n\,2\}\,(0b_1) &&= \mathit{as\text{-}izero}\\
&\mathit{iview}\,\{n\,2\}\,(i\,1_1)\ \textbf{\textit{with}}\ \mathit{iview}\,i\\
&\ldots \qquad\qquad\quad \mid \mathit{as\text{-}izero} &&= \mathit{as\text{-}isucc}\,0b_2\\
&\ldots \qquad\qquad\quad \mid \mathit{as\text{-}isucc}\,j &&= \mathit{as\text{-}isucc}\,(j\,2_2) \quad \text{-- borrow}\\
&\mathit{iview}\,\{n\,2\}\,(i\,2_1)\ \textbf{\textit{with}}\ \mathit{iview}\,i
\end{aligned}
$$

[5] Agda actually complains about a type mismatch as the two types are not *definitionally* equal. This somewhat unfortunate situation can, however, be fixed with a hint of extensionality, adding *succ-correct* to Agda's definitional equality.

$$
\begin{array}{lll}
\ldots & \mid \mathit{as\text{-}izero} & = \mathit{as\text{-}isucc}\,\mathtt{1}b_2 \\
\ldots & \mid \mathit{as\text{-}isucc}\,j & = \mathit{as\text{-}isucc}\,(j\,\mathtt{3}_2) \quad \text{-- borrow}
\end{array}
$$

Recall that the Peano view combines the test for zero and the predecessor function. The same is true of *iview*, except that arguments and results additionally alternate between the two number systems.

Finally, given the semantics of view patterns we can assert that *iview* does not change the value of its argument.

$$
\begin{array}{l}
\mathscr{W}[\![\_]\!] : \forall\,\{\,i : \mathit{Index}\,(\mathit{succ}\,n)\,\} \to \mathit{Index\text{-}View}\,i \to \mathit{Peano.Index}\,([\![\,n\,]\!]+1) \\
\mathscr{W}[\![\ \mathit{as\text{-}izero}\ \ ]\!] = \mathit{iz} \\
\mathscr{W}[\![\ \mathit{as\text{-}isucc}\,i\ ]\!] = \mathit{is}\,\mathscr{I}[\![\,i\,]\!] \\
\mathit{iview\text{-}correct} : (i : \mathit{Index}\,(\mathit{succ}\,n)) \to \mathscr{W}[\![\ \mathit{iview}\,i\ ]\!] \equiv \mathscr{I}[\![\,i\,]\!]
\end{array}
$$

7 Functions as Datastructures

To showcase the use of our new gadgets we adapt the implementation of Section 4 to binary indices, setting $\mathit{Array}\,n\,\mathit{elem} = \mathit{Index}\,n \to \mathit{elem}$.

$$
\begin{array}{l}
\mathit{nil} : \mathit{Array}\,\mathtt{0}b\,\mathit{elem} \\
\mathit{nil}\,() \\
\mathit{cons} : \mathit{elem} \to \mathit{Array}\,n\,\mathit{elem} \to \mathit{Array}\,(\mathit{succ}\,n)\,\mathit{elem} \\
\mathit{cons}\,x\,\mathit{xs}\,i\ \textbf{\textit{with}}\ \mathit{iview}\,i \\
\ldots \mid \mathit{as\text{-}izero} = x \\
\ldots \mid \mathit{as\text{-}isucc}\,j = \mathit{xs}\,j \\
\mathit{head} : \mathit{Array}\,(\mathit{succ}\,n)\,\mathit{elem} \to \mathit{elem} \\
\mathit{head}\,\mathit{xs} = \mathit{xs}\,\mathit{izero} \\
\mathit{tail} : \mathit{Array}\,(\mathit{succ}\,n)\,\mathit{elem} \to \mathit{Array}\,n\,\mathit{elem} \\
\mathit{tail}\,\mathit{xs}\,i = \mathit{xs}\,(\mathit{isucc}\,i)
\end{array}
$$

As in the Peano case, "functions as datastructures" serve as our reference implementation for datastructures based on Leibniz numerals. With this specification in place, we can now try to discover a corresponding *datastructure*.

8 One-two Trees

Turning to the heart of the matter, let us trieify the type of finite maps based on binary indices.

$$\mathit{trieify} : \forall\,\mathit{elem} \to \forall\,n \to (\mathit{Index}\,n \to \mathit{elem}) \cong \mathit{Array}\,n\,\mathit{elem}$$

The strategy should be clear: as in Section 5, we eliminate the type of finite maps using the laws of exponents. The base case is identical to the one for lists.

$$
\begin{array}{l}
\mathit{trieify}\ \mathit{elem}\ (\mathtt{0}b)\ = \\
\quad \textbf{\textit{proof}} \\
\qquad (\mathit{Index}\ (\mathtt{0}b)\ \to\ \mathit{elem}) \\
\quad \cong\langle\ \mathit{Index}\text{-}\mathtt{0}\ \cong\to\cong\ \cong\text{-}\mathit{reflexive}\ \rangle \\
\qquad (\bot\ \to\ \mathit{elem}) \\
\quad \cong\langle\ \mathit{law}\text{-}\mathit{of}\text{-}\mathit{exponents}\text{-}\bot\ \rangle \\
\qquad \top \\
\quad \cong\langle\ \mathit{use}\text{-}\mathit{as}\text{-}\mathit{definition}\text{-}\mathit{of}\ \mathit{Array}\text{-}\mathtt{0}\ \rangle \\
\qquad \mathit{Array}\ (\mathtt{0}b)\ \mathit{elem} \\
\quad \blacksquare
\end{array}
$$

The calculation for the inductive cases follows the same rhythm—we unfold the definition of *Index* and apply the laws of exponents—except that we additionally invoke the induction hypothesis.

$$
\begin{array}{l}
\mathit{trieify}\ \mathit{elem}\ (n\,\mathtt{1})\ = \\
\quad \textbf{\textit{proof}} \\
\qquad (\mathit{Index}\ (n\,\mathtt{1})\ \to\ \mathit{elem}) \\
\quad \cong\langle\ \mathit{Index}\text{-}\mathtt{1}\ \cong\to\cong\ \cong\text{-}\mathit{reflexive}\ \rangle \\
\qquad (\top \uplus \mathit{Index}\ n \uplus \mathit{Index}\ n\ \to\ \mathit{elem}) \\
\quad \cong\langle\ \cong\text{-}\mathit{transitive}\ \mathit{law}\text{-}\mathit{of}\text{-}\mathit{exponents}\text{-}\uplus\ (\mathit{law}\text{-}\mathit{of}\text{-}\mathit{exponents}\text{-}\top\ \cong\times\cong\ \mathit{law}\text{-}\mathit{of}\text{-}\mathit{exponents}\text{-}\uplus)\ \rangle \\
\qquad \mathit{elem} \times (\mathit{Index}\ n\ \to\ \mathit{elem}) \times (\mathit{Index}\ n\ \to\ \mathit{elem}) \\
\quad \cong\langle\ (\cong\text{-}\mathit{reflexive}\ \cong\times\cong\ (\mathit{trieify}\ \mathit{elem}\ n\ \cong\times\cong\ \mathit{trieify}\ \mathit{elem}\ n))\ \rangle \\
\qquad \mathit{elem} \times \mathit{Array}\ n\ \mathit{elem} \times \mathit{Array}\ n\ \mathit{elem} \\
\quad \cong\langle\ \mathit{use}\text{-}\mathit{as}\text{-}\mathit{definition}\text{-}\mathit{of}\ \mathit{Array}\text{-}\mathtt{1}\ \rangle \\
\qquad \mathit{Array}\ (n\,\mathtt{1})\ \mathit{elem} \\
\quad \blacksquare
\end{array}
$$

The final step in the isomorphism above expresses that an array of size $n \cdot 2 + 1$ consists of an element followed by two arrays of size n. The isomorphism for arrays of size $n \cdot 2 + 2$ follows a similar pattern.

$$
\begin{array}{l}
\mathit{trieify}\ \mathit{elem}\ (n\,\mathtt{2})\ = \\
\quad \textbf{\textit{proof}} \\
\qquad (\mathit{Index}\ (n\,\mathtt{2})\ \to\ \mathit{elem}) \\
\quad \cong\langle\ \mathit{Index}\text{-}\mathtt{2}\ \cong\to\cong\ \cong\text{-}\mathit{reflexive}\ \rangle \\
\qquad (\top \uplus \top \uplus \mathit{Index}\ n \uplus \mathit{Index}\ n\ \to\ \mathit{elem}) \\
\quad \cong\langle\ \cong\text{-}\mathit{transitive}\ \mathit{law}\text{-}\mathit{of}\text{-}\mathit{exponents}\text{-}\uplus\ (\mathit{law}\text{-}\mathit{of}\text{-}\mathit{exponents}\text{-}\top\ \cong\times\cong\ \mathit{law}\text{-}\mathit{of}\text{-}\mathit{exponents}\text{-}\uplus)\ \rangle \\
\qquad \mathit{elem} \times (\top \to \mathit{elem}) \times (\mathit{Index}\ n \uplus \mathit{Index}\ n\ \to\ \mathit{elem}) \\
\quad \cong\langle\ \cong\text{-}\mathit{reflexive}\ \cong\times\cong\ (\mathit{law}\text{-}\mathit{of}\text{-}\mathit{exponents}\text{-}\top\ \cong\times\cong\ \mathit{law}\text{-}\mathit{of}\text{-}\mathit{exponents}\text{-}\uplus)\ \rangle \\
\qquad \mathit{elem} \times (\mathit{elem} \times (\mathit{Index}\ n\ \to\ \mathit{elem}) \times (\mathit{Index}\ n\ \to\ \mathit{elem})) \\
\quad \cong\langle\ \cong\text{-}\mathit{reflexive}\ \cong\times\cong\ (\cong\text{-}\mathit{reflexive}\ \cong\times\cong\ (\mathit{trieify}\ \mathit{elem}\ n\ \cong\times\cong\ \mathit{trieify}\ \mathit{elem}\ n))\ \rangle \\
\qquad \mathit{elem} \times \mathit{elem} \times \mathit{Array}\ n\ \mathit{elem} \times \mathit{Array}\ n\ \mathit{elem} \\
\quad \cong\langle\ \mathit{use}\text{-}\mathit{as}\text{-}\mathit{definition}\text{-}\mathit{of}\ \mathit{Array}\text{-}\mathtt{2}\ \rangle \\
\qquad \mathit{Array}\ (n\,\mathtt{2})\ \mathit{elem} \\
\quad \blacksquare
\end{array}
$$

An array of size $n \cdot 2 + 2$ consists of two elements followed by two arrays of size n. All in all, we obtain the following datatype. Its elements are called *one-two trees*[6] for want of a better name.

$$
\begin{aligned}
&\textbf{\textit{variable}}\ \mathit{elem} : \mathit{Set} \\
&\textbf{\textit{data}}\ \mathit{Array} : \mathit{Leibniz} \to \mathit{Set} \to \mathit{Set}\ \textbf{\textit{where}} \\
&\quad \mathit{Leaf} \quad : \mathit{Array}\ \mathtt{0}b\ \mathit{elem} \\
&\quad \mathit{Node}_1 : \mathit{elem} \to \qquad \mathit{Array}\ n\ \mathit{elem} \to \mathit{Array}\ n\ \mathit{elem} \to \mathit{Array}\ (n\,\mathtt{1})\ \mathit{elem} \\
&\quad \mathit{Node}_2 : \mathit{elem} \to \mathit{elem} \to \mathit{Array}\ n\ \mathit{elem} \to \mathit{Array}\ n\ \mathit{elem} \to \mathit{Array}\ (n\,\mathtt{2})\ \mathit{elem}
\end{aligned}
$$

As an aside, Agda like Haskell prefers curried data constructors over uncurried ones. The following equivalent definition that uses pairs shows more clearly that one-two trees are modelled after the `1-2` number system,

$$
\begin{aligned}
&\textbf{\textit{data}}\ \mathit{Array}' : \mathit{Leibniz} \to \mathit{Set} \to \mathit{Set}\ \textbf{\textit{where}} \\
&\quad \mathit{Leaf} \quad : \mathit{Array}'\ \mathtt{0}b\ \mathit{elem} \\
&\quad \mathit{Node}_1 : \mathit{elem}^{\,1} \to (\mathit{Array}'\ n\ \mathit{elem})^{\,2} \to \mathit{Array}'\ (n\,\mathtt{1})\ \mathit{elem} \\
&\quad \mathit{Node}_2 : \mathit{elem}^{\,2} \to (\mathit{Array}'\ n\ \mathit{elem})^{\,2} \to \mathit{Array}'\ (n\,\mathtt{2})\ \mathit{elem}
\end{aligned}
$$

where $A^{\,1} = A$ and $A^{\,2} = A \times A$.

Turning to the operations on one-two trees, we first extract the witnesses of the *trieify* isomorphism, obtaining human-readable definitions of *lookup* and *tabulate*.

$$
\begin{aligned}
&\mathit{lookup} : \mathit{Array}\ n\ \mathit{elem} \to (\mathit{Index}\ n \to \mathit{elem}) \\
&\mathit{lookup}\ (\mathit{Node}_1\ x_0 \quad l\ r)\ (\mathtt{0}b_1) \ = \ x_0 \\
&\mathit{lookup}\ (\mathit{Node}_1\ x_0 \quad l\ r)\ (i\,\mathtt{1}_1) \ = \ \mathit{lookup}\ l\ i \\
&\mathit{lookup}\ (\mathit{Node}_1\ x_0 \quad l\ r)\ (i\,\mathtt{2}_1) \ = \ \mathit{lookup}\ r\ i \\
&\mathit{lookup}\ (\mathit{Node}_2\ x_0\ x_1\ l\ r)\ (\mathtt{0}b_2) \ = \ x_0 \\
&\mathit{lookup}\ (\mathit{Node}_2\ x_0\ x_1\ l\ r)\ (\mathtt{1}b_2) \ = \ x_1 \\
&\mathit{lookup}\ (\mathit{Node}_2\ x_0\ x_1\ l\ r)\ (i\,\mathtt{2}_2) \ = \ \mathit{lookup}\ l\ i \\
&\mathit{lookup}\ (\mathit{Node}_2\ x_0\ x_1\ l\ r)\ (i\,\mathtt{3}_2) \ = \ \mathit{lookup}\ r\ i
\end{aligned}
$$

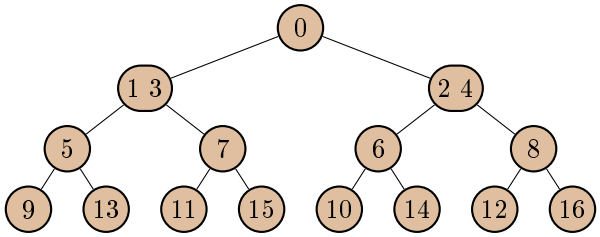
The implementation of *lookup* nicely illustrates the central idea of tries, where the index serves as a path into the tree. The least significant digit selects the node component. If the component is a sub-tree, then *lookup* recurses. The diagrams below visualize the indexing scheme.

[6] One-two trees are unconnected with 1-2 brother trees, an implementation of AVL trees.

A one-node corresponds to the digit `1`, a two-node corresponds to `2`. Conversely, the *tabulate* function computes the array corresponding to a given finite map.

$$
\begin{array}{l}
\mathit{tabulate} : (\mathit{Index}\, n \to \mathit{elem}) \to \mathit{Array}\, n\, \mathit{elem} \\
\mathit{tabulate}\, \{\mathtt{0}b\}\ f = \mathit{Leaf} \\
\mathit{tabulate}\, \{n\, \mathtt{1}\}\, f = \mathit{Node}_1\, (f\, \mathtt{0}b_1) \qquad\quad (\mathit{tabulate}\, (\lambda\, i \to f\, (i\, \mathtt{1}_1))) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad\ (\mathit{tabulate}\, (\lambda\, i \to f\, (i\, \mathtt{2}_1))) \\
\mathit{tabulate}\, \{n\, \mathtt{2}\}\, f = \mathit{Node}_2\, (f\, \mathtt{0}b_2)\, (f\, \mathtt{1}b_2)\, (\mathit{tabulate}\, (\lambda\, i \to f\, (i\, \mathtt{2}_2))) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad\ (\mathit{tabulate}\, (\lambda\, i \to f\, (i\, \mathtt{3}_2)))
\end{array}
$$

For example, $\mathit{tabulate}\, \{\mathtt{0}b\, \mathtt{1}\, \mathtt{1}\, \mathtt{2}\, \mathtt{1}\}\, \mathit{id}$ yields the tree depicted below.



A couple of remarks are in order. By definition, one-two trees are not only *size-balanced*, they are also *height-balanced*—the height corresponds to the length of the binary representation of the size. The binary decomposition of the size fully determines the shape of the tree; all the nodes on one level have the same "shape"; the digits determine this shape from bottom to top. In the example above, $\mathtt{0}b\, \mathtt{1}\, \mathtt{1}\, \mathtt{2}\, \mathtt{1}$ implies that the nodes on the third level (from bottom to top) are two-nodes, whereas the other nodes are one-nodes. There are 2^3 nodes on the bottom level, witnessing the weight of the most significant digit.

Turning to the size-changing operations, *cons* is based on the binary increment. Recall that *succ* alternates between odd and even numbers. Accordingly, *cons* alternates between one- and two-nodes.

$$
\begin{array}{l}
\mathit{nil} : \mathit{Array}\, \mathit{zero}\, \mathit{elem} \\
\mathit{nil} = \mathit{Leaf} \\
\mathit{cons} : \mathit{elem} \to \mathit{Array}\, n\, \mathit{elem} \to \mathit{Array}\, (\mathit{succ}\, n)\, \mathit{elem} \\
\mathit{cons}\, x_0\, (\mathit{Leaf}) \qquad\quad = \mathit{Node}_1\, x_0\, \mathit{Leaf}\, \mathit{Leaf} \\
\mathit{cons}\, x_0\, (\mathit{Node}_1\, x_1 \quad l\, r) = \mathit{Node}_2\, x_0\, x_1\, l\, r \\
\mathit{cons}\, x_0\, (\mathit{Node}_2\, x_1\, x_2\, l\, r) = \mathit{Node}_1\, x_0\, (\mathit{cons}\, x_1\, l)\, (\mathit{cons}\, x_2\, r)
\end{array}
$$

A one-node is turned into a two-node. Dually, a two-node becomes a one-node; the two surplus elements are pushed into the two sub-trees. Observe that the recursion pattern of *succ* dictates the recursion pattern of *cons*, that is, whether we stop or recurse. The definition of *isucc* dictates the layout of the data. For example, the first component of Node_1 becomes the second component of Node_2. You may want to view a two-node as a small buffer. Consing an element to a one-node allocates the buffer; consing a further element causes the buffer to overflow.

It is also possible to *derive* the implementation of *cons* from the specification (2a)–(2b). However, as the argument is based on positions and the *Index* type comprises seven constructors, the calculations are rather lengthy, but wholly unsurprising, so they have been relegated to Appendix A.

Figure 1 shows a succession of one-two trees obtained by consing 4, 3, 2, 1, and 0 (in this order) to $tabulate\,\{0b\,1\,2\,2\}\,(\lambda\,i\;\rightarrow\;i+5)$. In every second step, *cons* touches only the root node. However, every once in a while the entire tree is rewritten, corresponding to a cascading carry. In Figure 1, this happens in the final step when a tree of size $0b\,2\,2\,2$ is turned into a tree of size $0b\,1\,1\,1\,1$. Consequently, the *worst-case* running time of *cons* is $\Theta(n)$. However, like the

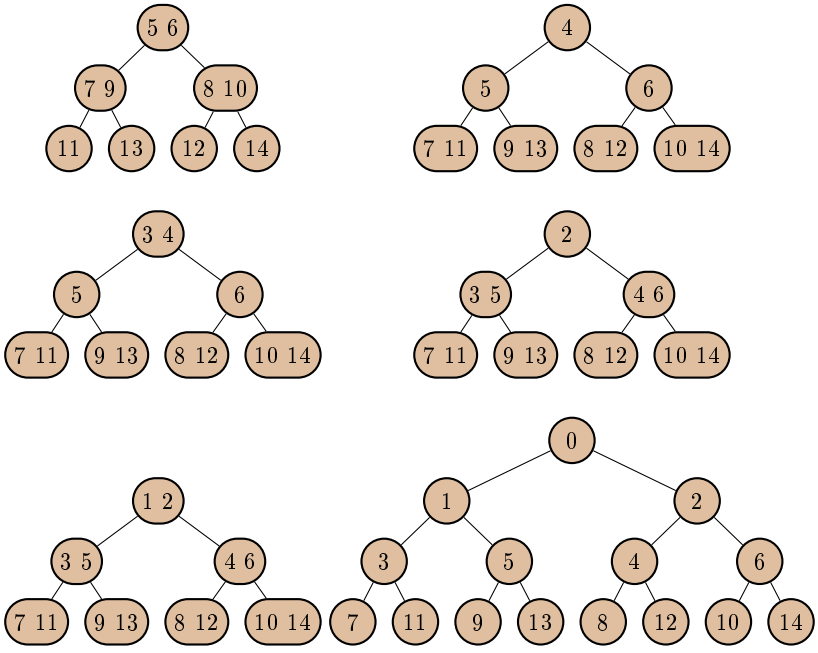


Fig. 1. One-two trees of shape $0b\,1\,2\,2$ up to $0b\,1\,1\,1\,1$.

binary increment, consing shows a more favourable behaviour if a sequence of operations is taken into account: *cons* runs in $\Theta(\log n)$ *amortized* time. This is less favourable than *succ*, which runs in constant amortized time. The reason is simple: for each carry *succ* makes one recursive call, whereas *cons* features two calls so a carry propagation takes time proportional to the weight of the digit. We return to this point in Section 10.

The operations *head* and *tail* basically undo the changes of *cons*.

$$\begin{aligned} &head\,:\,Array\,(succ\,n)\,elem\;\rightarrow\;elem\\ &head\,\{0b\}\;\;(Node_1\,x_0\quad\;\;l\,r)\;=\;x_0 \end{aligned}$$

$$
\begin{aligned}
&\mathit{head}\,\{n\,\mathtt{1}\}\,(\mathit{Node}_2\;x_0\;x_1\;l\;r) &&= x_0\\
&\mathit{head}\,\{n\,\mathtt{2}\}\,(\mathit{Node}_1\;x_0\quad\;\;l\;r) &&= x_0\\
&\mathit{tail} : \mathit{Array}\,(\mathit{succ}\,n)\;\mathit{elem} \to \mathit{Array}\,n\;\mathit{elem}\\
&\mathit{tail}\,\{\mathtt{0}b\}\;\;(\mathit{Node}_1\;x_0\quad\;\;l\;r) &&= \mathit{Leaf}\\
&\mathit{tail}\,\{n\,\mathtt{1}\}\,(\mathit{Node}_2\;x_0\;x_1\;l\;r) &&= \mathit{Node}_1\;x_1\;l\;r\\
&\mathit{tail}\,\{n\,\mathtt{2}\}\,(\mathit{Node}_1\;x_0\quad\;\;l\;r) &&= \mathit{Node}_2\;(\mathit{head}\,l)\;(\mathit{head}\,r)\;(\mathit{tail}\,l)\;(\mathit{tail}\,r)
\end{aligned}
$$

As an attractive alternative to these operations we also introduce a list view, analogous to the Peano view on binary numbers.

$$
\begin{aligned}
&\textbf{\textit{data}}\;\mathit{List\text{-}View} : \mathit{Peano\text{-}View}\,n \to \mathit{Set} \to \mathit{Set}\;\textbf{\textit{where}}\\
&\quad \mathit{as\text{-}nil}\quad\;\; : \mathit{List\text{-}View}\;\mathit{as\text{-}zero}\;\mathit{elem}\\
&\quad \mathit{as\text{-}cons} : \mathit{elem} \to \mathit{Array}\,n\;\mathit{elem} \to \mathit{List\text{-}View}\,(\mathit{as\text{-}succ}\,n)\;\mathit{elem}
\end{aligned}
$$

Observe that the list view is indexed by the Peano view. The view function itself illustrates the use of view patterns.

$$
\begin{aligned}
&\mathit{list\text{-}view} : \{n : \mathit{Leibniz}\} \to \mathit{Array}\,n\;\mathit{elem} \to \mathit{List\text{-}View}\,(\mathit{view}\,n)\;\mathit{elem}\\
&\mathit{list\text{-}view}\,\{n = \mathtt{0}b\}\;\;(\mathit{Leaf}) &&= \mathit{as\text{-}nil}\\
&\mathit{list\text{-}view}\,\{n = n\,\mathtt{1}\}\,(\mathit{Node}_1\;x_0\;l\;r)\;\textbf{\textit{with}}\;\mathit{view}\,n \mid \mathit{list\text{-}view}\,l \mid \mathit{list\text{-}view}\,r\\
&\ldots \mid \mathit{as\text{-}zero}\quad\;\; \mid \mathit{as\text{-}nil}\quad\;\; \mid \mathit{as\text{-}nil}\quad\;\; &&= \mathit{as\text{-}cons}\;x_0\;r\\
&\ldots \mid \mathit{as\text{-}succ}\,m \mid \mathit{as\text{-}cons}\;x_1\;l' \mid \mathit{as\text{-}cons}\;x_2\;r' &&= \mathit{as\text{-}cons}\;x_0\;(\mathit{Node}_2\;x_1\;x_2\;l'\;r')\\
&\mathit{list\text{-}view}\,\{n = n\,\mathtt{2}\}\,(\mathit{Node}_2\;x_0\;x_1\;l\;r) &&= \mathit{as\text{-}cons}\;x_0\;(\mathit{Node}_1\;x_1\;l\;r)
\end{aligned}
$$

The first and the last equation are straightforward, in particular, removing an element from a two-node yields a one-node. Following this logic, removing an element from a one-node gives a zero-node—except that our datatype does not feature this node type. Consequently, we need to borrow data from the sub-trees. To this end, a view is simultaneously placed on the size and the two sub-trees—a typical usage pattern.

The implementation of arrays using one-two trees can be shown correct with respect to our reference implementation in Section 7. The steps are completely analogous to the line of action in Section 5. The details are elided for reasons of space.

Remark 5 (Haskell). It is instructive to translate the Agda code into a language such as Haskell that does not support dependent types. The datatype definition is almost the same, except that the size index is dropped.

$$
\begin{aligned}
&\textbf{\textit{data}}\;\mathit{Array} :: \mathit{Type} \to \mathit{Type}\;\textbf{\textit{where}}\\
&\quad \mathit{Leaf}\quad\;\; :: \mathit{Array}\;\mathit{elem}\\
&\quad \mathit{Node}_1 :: \mathit{elem} \to \qquad\;\; \mathit{Array}\;\mathit{elem} \to \mathit{Array}\;\mathit{elem} \to \mathit{Array}\;\mathit{elem}\\
&\quad \mathit{Node}_2 :: \mathit{elem} \to \mathit{elem} \to \mathit{Array}\;\mathit{elem} \to \mathit{Array}\;\mathit{elem} \to \mathit{Array}\;\mathit{elem}
\end{aligned}
$$

If we represent the element of the index type by plain integers, then we need to translate the index patterns. This can be done in a fairly straightforward manner using guards and integer division.

$$
\begin{array}{lll}
\mathit{lookup} :: \mathit{Array}\,\mathit{elem} \to (\mathit{Integer} \to \mathit{elem}) & & \\
\mathit{lookup}\,(\mathit{Node}_1\,x_0 \quad l\,r)\,i & \mid i \equiv 0 & = x_0 \\
& \mid i\ \text{`}\mathit{mod}\text{`}\ 2 \equiv 1 & = \mathit{lookup}\,l\,((i\text{-}1)\ \text{`}\mathit{div}\text{`}\ 2) \\
& \mid i\ \text{`}\mathit{mod}\text{`}\ 2 \equiv 0 & = \mathit{lookup}\,r\,((i\text{-}2)\ \text{`}\mathit{div}\text{`}\ 2) \\
\mathit{lookup}\,(\mathit{Node}_2\,x_0\,x_1\,l\,r)\,i & \mid i \equiv 0 & = x_0 \\
& \mid i \equiv 1 & = x_1 \\
& \mid i\ \text{`}\mathit{mod}\text{`}\ 2 \equiv 0 & = \mathit{lookup}\,l\,((i\text{-}2)\ \text{`}\mathit{div}\text{`}\ 2) \\
& \mid i\ \text{`}\mathit{mod}\text{`}\ 2 \equiv 1 & = \mathit{lookup}\,r\,((i\text{-}3)\ \text{`}\mathit{div}\text{`}\ 2)
\end{array}
$$

A more sophisticated alternative is to replace each constructor of *Index* by a *pattern synonym* [23].

As the interface considered in this paper is rather narrow, there is no need to maintain the size of trees at run-time. However, if size information is needed, it can be computed on the fly,

$$
\begin{array}{lcl}
\mathit{size} :: \mathit{Array}\,\mathit{elem} \to \mathit{Integer} & & \\
\mathit{size}\,(\mathit{Leaf}) & = & 0 \\
\mathit{size}\,(\mathit{Node}_1\,x_1 \quad l\,r) & = & \mathit{size}\,l * 2 + 1 \\
\mathit{size}\,(\mathit{Node}_2\,x_1\,x_2\,l\,r) & = & \mathit{size}\,l * 2 + 2
\end{array}
$$

in logarithmic time. □

9 Braun Trees

The derivation of the Leibniz index type in Section 6.2 and its associated trie type in Section 8 are entirely straightforward. Too straightforward, perhaps? This section and the next highlight the decision points and investigate alternative designs.

9.1 Index Type, Revisited

The index type for *Peano numerals* enjoys an appealing property: its constructors look and behave like their Peano namesakes, indicated by the isomorphisms:

$$\mathit{Peano} \cong \top \uplus \mathit{Peano} \qquad \mathit{Index}\,(\mathit{succ}\,n) \cong \top \uplus \mathit{Index}\,(n)\ .$$

The same cannot be said of Leibniz indices. The indices below an even bound are based on 2-3 binary numbers, rather than the 1-2 system we started with.

$$\mathit{Leibniz} \cong \top \uplus \mathit{Leibniz} \uplus \mathit{Leibniz} \quad \mathit{Index}\,(n\,2) \cong \top \uplus \top \uplus \mathit{Index}\,(n) \uplus \mathit{Index}\,(n)$$

Can we re-work the isomorphism on the right so that it has the *same shape* as the one on the left, with three constructors instead of four?

Let's calculate, revisiting the second inductive case.

$$
\begin{array}{l}
\mathit{re\text{-}index}\,(n\,2) = \\
\quad \textbf{\textit{proof}}
\end{array}
$$

$$
\begin{array}{l}
\quad \mathit{Peano.Index}\,[\![\, n\,2 \,]\!] \\
\cong\langle\ \cong\text{-}\mathit{transitive}\ \mathit{Index\text{-}succ}\ (\cong\text{-}\mathit{reflexive} \uplus\cong \mathit{Index\text{-}succ})\ \rangle \\
\quad \top \uplus \top \uplus \mathit{Peano.Index}\,([\![\, n \,]\!] \cdot 2) \\
\cong\langle\ \cong\text{-}\mathit{reflexive} \uplus\cong (\cong\text{-}\mathit{reflexive} \uplus\cong \mathit{Index\text{-}2{\cdot}n} \uplus\cong \mathit{Index\text{-}n} \uplus \mathit{Index\text{-}n})\ \rangle \\
\quad \top \uplus \top \uplus \mathit{Peano.Index}\,[\![\, n \,]\!] \uplus \mathit{Peano.Index}\,[\![\, n \,]\!]
\end{array}
$$

At this point, we have applied the induction hypothesis in the original derivation. An alternative is to first join the second and third summands of the disjoint union, applying the re-indexing law *Index-succ* backwards from right to left.

$$
\begin{array}{l}
\cong\langle\ \cong\text{-}\mathit{reflexive} \uplus\cong \cong\text{-}\mathit{symmetric}\ \uplus\text{-}\mathit{associative}\ \rangle \\
\quad \top \uplus (\top \uplus \mathit{Peano.Index}\,[\![\, n \,]\!]) \uplus \mathit{Peano.Index}\,[\![\, n \,]\!] \\
\cong\langle\ \cong\text{-}\mathit{reflexive} \uplus\cong (\cong\text{-}\mathit{symmetric}\ \mathit{Index\text{-}succ} \uplus\cong \cong\text{-}\mathit{reflexive})\ \rangle \\
\quad \top \uplus \mathit{Peano.Index}\,(\mathit{Peano.succ}\,[\![\, n \,]\!]) \uplus \mathit{Peano.Index}\,[\![\, n \,]\!] \\
\cong\langle\ (\cong\text{-}\mathit{reflexive} \uplus\cong (\cong\text{-}\mathit{congruence}\ \mathit{Peano.Index}\,(\mathit{symmetric}\ \mathit{succ\text{-}correct}) \uplus\cong \cong\text{-}\mathit{reflexive}))\ \rangle \\
\quad \top \uplus \mathit{Peano.Index}\,[\![\, \mathit{Leibniz.succ}\ n \,]\!] \uplus \mathit{Peano.Index}\,[\![\, n \,]\!] \\
\cong\langle\ \cong\text{-}\mathit{reflexive} \uplus\cong (\mathit{re\text{-}index}\,(\mathit{succ}\ n) \uplus\cong \mathit{re\text{-}index}\ n)\ \rangle \\
\quad \top \uplus \mathit{Leibniz.Index}\,(\mathit{Leibniz.succ}\ n) \uplus \mathit{Leibniz.Index}\ n \\
\cong\langle\ \mathit{use\text{-}as\text{-}definition\text{-}of}\ \mathit{Index}\text{-}2\ \rangle \\
\quad \mathit{Leibniz.Index}\,(n\,2) \\
\blacksquare
\end{array}
$$

Voilà! Naming the anonymous summands, we arrive at the following, alternative index type for Leibniz numerals.

$$
\begin{array}{l}
\textbf{\textit{data}}\ \mathit{Index}\ :\ \mathit{Leibniz} \to \mathit{Set}\ \textbf{\textit{where}} \\
\quad \mathtt{0}b_1\ :\ \qquad\qquad\quad \mathit{Index}\,(n\,\mathtt{1}) \\
\quad \_\mathtt{1}_1\ :\ \mathit{Index}\ n \quad \to \mathit{Index}\,(n\,\mathtt{1}) \\
\quad \_\mathtt{2}_1\ :\ \mathit{Index}\ n \quad \to \mathit{Index}\,(n\,\mathtt{1}) \\
\quad \mathtt{0}b_2\ :\ \qquad\qquad\quad \mathit{Index}\,(n\,\mathtt{2}) \\
\quad \_\mathtt{1}_2\ :\ \mathit{Index}\,(\mathit{succ}\ n) \to \mathit{Index}\,(n\,\mathtt{2}) \\
\quad \_\mathtt{2}_2\ :\ \mathit{Index}\ n \quad \to \mathit{Index}\,(n\,\mathtt{2})
\end{array}
$$

The datatype features two identical sets of constructors, one for indices below an odd upper bound and a second for indices below an even upper bound.

Having changed the index type, we need to adapt the operations on indices.

$$
\begin{array}{l}
\mathit{izero}\ :\ \mathit{Index}\,(\mathit{succ}\ n) \\
\mathit{izero}\,\{\mathtt{0}b\}\ =\ \mathtt{0}b_1 \\
\mathit{izero}\,\{n\,\mathtt{1}\}\ =\ \mathtt{0}b_2 \\
\mathit{izero}\,\{n\,\mathtt{2}\}\ =\ \mathtt{0}b_1 \\
\\
\mathit{isucc}\ :\ \mathit{Index}\ n \to \mathit{Index}\,(\mathit{succ}\ n) \\
\mathit{isucc}\,\{n\,\mathtt{1}\}\,(\mathtt{0}b_1)\ =\ \mathit{izero}\,\mathtt{1}_2 \\
\mathit{isucc}\,\{n\,\mathtt{1}\}\,(i\,\mathtt{1}_1)\ =\ i\,\mathtt{2}_2 \\
\mathit{isucc}\,\{n\,\mathtt{1}\}\,(i\,\mathtt{2}_1)\ =\ (\mathit{isucc}\ i)\,\mathtt{1}_2 \\
\mathit{isucc}\,\{n\,\mathtt{2}\}\,(\mathtt{0}b_2)\ =\ \mathit{izero}\,\mathtt{1}_1 \\
\mathit{isucc}\,\{n\,\mathtt{2}\}\,(i\,\mathtt{1}_2)\ =\ i\,\mathtt{2}_1 \\
\mathit{isucc}\,\{n\,\mathtt{2}\}\,(i\,\mathtt{2}_2)\ =\ (\mathit{isucc}\ i)\,\mathtt{1}_1
\end{array}
$$

If we ignore the subscripts, the first three clauses of the successor function are identical to the last three clauses. Operationally, the constructors 1_1 and 1_2 are treated in exactly the same way. This is precisely what we have hoped for! (At the risk of dwelling on the obvious, even though the definition of *isucc* seems repetitive, it is not: the proofs relating the indices to their upper bounds are quite different.)

9.2 Trie Type, Revisited

The new index type gives rise to a new trie type. We only need to adapt the trieification for the second inductive case: $n\,2$. As in the original derivation, the steps are entirely straightforward—nothing surprising here.

$$
\begin{array}{l}
\mathit{trieify}\ \mathit{elem}\ (n\,2)\ = \\
\quad \textbf{\textit{proof}} \\
\qquad (\mathit{Index}\ (n\,2)\ \to\ \mathit{elem}) \\
\quad \cong\langle\ \mathit{Index}\text{-}2\ \cong\to\cong\ \cong\text{-}\mathit{reflexive}\ \rangle \\
\qquad (\top \uplus \mathit{Index}\ (\mathit{succ}\ n) \uplus \mathit{Index}\ n\ \to\ \mathit{elem}) \\
\quad \cong\langle\ \cong\text{-}\mathit{transitive}\ \mathit{law\text{-}of\text{-}exponents}\text{-}\uplus\ (\mathit{law\text{-}of\text{-}exponents}\text{-}\top\ \cong\times\cong\ \mathit{law\text{-}of\text{-}exponents}\text{-}\uplus)\ \rangle \\
\qquad \mathit{elem} \times (\mathit{Index}\ (\mathit{succ}\ n)\ \to\ \mathit{elem}) \times (\mathit{Index}\ n\ \to\ \mathit{elem}) \\
\quad \cong\langle\ (\cong\text{-}\mathit{reflexive}\ \cong\times\cong\ (\mathit{trieify}\ \mathit{elem}\ (\mathit{succ}\ n)\ \cong\times\cong\ \mathit{trieify}\ \mathit{elem}\ n))\ \rangle \\
\qquad \mathit{elem} \times \mathit{Array}\ (\mathit{succ}\ n)\ \mathit{elem} \times \mathit{Array}\ n\ \mathit{elem} \\
\quad \cong\langle\ \mathit{use\text{-}as\text{-}definition\text{-}of}\ \mathit{Array}\text{-}2\ \rangle \\
\qquad \mathit{Array}\ (n\,2)\ \mathit{elem} \\
\quad \blacksquare
\end{array}
$$

We obtain the following type of tries, where the subscript attached to the node constructors indicates the least significant digit of the upper bound (1 or 2).

$$
\begin{array}{l}
\textbf{\textit{data}}\ \mathit{Array}\ :\ \mathit{Leibniz} \to \mathit{Set} \to \mathit{Set}\ \textbf{\textit{where}} \\
\quad \mathit{Leaf}\ \ \ :\ \mathit{Array}\,0b\ \mathit{elem} \\
\quad \mathit{Node}_1\ :\ \mathit{elem} \to \mathit{Array}\ n \qquad\quad \mathit{elem} \to \mathit{Array}\ n\ \mathit{elem} \to \mathit{Array}\ (n\,1)\ \mathit{elem} \\
\quad \mathit{Node}_2\ :\ \mathit{elem} \to \mathit{Array}\ (\mathit{succ}\ n)\ \mathit{elem} \to \mathit{Array}\ n\ \mathit{elem} \to \mathit{Array}\ (n\,2)\ \mathit{elem}
\end{array}
$$

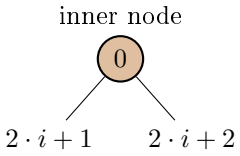
A moment's reflection reveals that we have rediscovered *Braun trees*. Recall that the size of a Braun tree determines its shape: a Braun tree of odd size $(n\,1)$ consists of two sub-trees of the same size; in a Braun tree of even, non-zero size $(n\,2)$ the left sub-tree is one element larger. As an aside, the property that the size determines the shape is shared by all our implementations of flexible arrays. It is a consequence of the fact that the container types are based on *non-redundant* number systems.

Similar to one-two threes, erm, trees, Braun trees feature two constructors for non-empty trees. However, in contrast to one-two trees, indexing is the same for both constructors. This becomes apparent if we extract the witnesses from the *trieify* ismorphism.

$$
\begin{array}{l}
\mathit{lookup}\ :\ \mathit{Array}\ n\ \mathit{elem} \to (\mathit{Index}\ n \to \mathit{elem}) \\
\mathit{lookup}\ (\mathit{Node}_1\ x_0\ l\ r)\ (0b_1)\ =\ x_0
\end{array}
$$

$$
\begin{aligned}
&\mathit{lookup}\,(\mathit{Node}_1\ x_0\ l\ r)\,(i\,\mathtt{1}_1) &&= \mathit{lookup}\ l\ i\\
&\mathit{lookup}\,(\mathit{Node}_1\ x_0\ l\ r)\,(i\,\mathtt{2}_1) &&= \mathit{lookup}\ r\ i\\
&\mathit{lookup}\,(\mathit{Node}_2\ x_0\ l\ r)\,(\mathtt{0}b_2) &&= x_0\\
&\mathit{lookup}\,(\mathit{Node}_2\ x_0\ l\ r)\,(i\,\mathtt{1}_2) &&= \mathit{lookup}\ l\ i\\
&\mathit{lookup}\,(\mathit{Node}_2\ x_0\ l\ r)\,(i\,\mathtt{2}_2) &&= \mathit{lookup}\ r\ i
\end{aligned}
$$

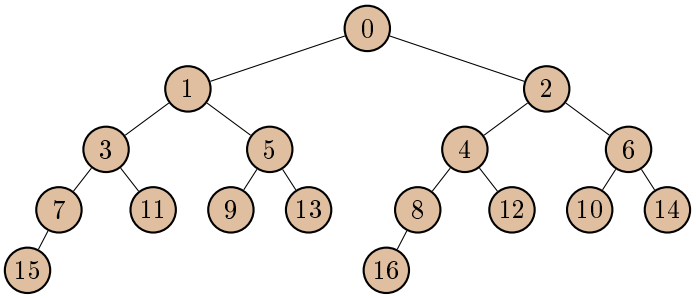
We make the same observation as for the successor function: if we ignore the subscripts, the first three clauses are identical to the last three clauses. In other words, the same indexing scheme applies to both varieties of inner nodes:



The definition of *tabulate* is similarly repetitive.

$$
\begin{aligned}
&\mathit{tabulate} : (\mathit{Index}\ n \to \mathit{elem}) \to \mathit{Array}\ n\ \mathit{elem}\\
&\mathit{tabulate}\,\{\mathtt{0}b\}\ f &&= \mathit{Leaf}\\
&\mathit{tabulate}\,\{n\,\mathtt{1}\}\,f &&= \mathit{Node}_1\ (f\,\mathtt{0}b_1)\ (\mathit{tabulate}\,(\lambda\, i \to f(i\,\mathtt{1}_1)))\ (\mathit{tabulate}\,(\lambda\, i \to f(i\,\mathtt{2}_1)))\\
&\mathit{tabulate}\,\{n\,\mathtt{2}\}\,f &&= \mathit{Node}_2\ (f\,\mathtt{0}b_2)\ (\mathit{tabulate}\,(\lambda\, i \to f(i\,\mathtt{1}_2)))\ (\mathit{tabulate}\,(\lambda\, i \to f(i\,\mathtt{2}_2)))
\end{aligned}
$$

For example, the call $\mathit{tabulate}\,\{\mathtt{0}b\,\mathtt{1}\,\mathtt{1}\,\mathtt{2}\,\mathtt{1}\}\ \mathit{id}$ yields the Braun tree shown below.



Here we observe the effect of the re-indexing isomorphism based on zipping or interleaving, see Section 6.2. Elements at odd positions are located in the left sub-tree, elements at even, non-zero positions in the right sub-tree.

There is, however, one problem with this definition. The Node_2 constructor has two subtrees: one of size n, the other of size $\mathit{succ}\ n$. As a result, the (implicit) Leibniz number passed implicitly to the *tabulate* function is not obviously decreasing: one call is passed n; the other is passed $\mathit{succ}\ n$. While the latter represents a smaller natural number, it is not a structurally smaller recursive call. As a result, Agda rejects this definition as it stands. There is a reasonably straightforward argument that we can make to guarantee termination—even if the recursion is not structural, it is well-founded: each recursive call is performed on a structurally smaller Peano number. In the remainder of this section, we will ignore such termination issues.

As before, the indexing scheme determines the implementation of the size-changing operations.

$$
\begin{aligned}
&\mathit{nil} : \mathit{Array}\ \mathit{zero}\ \mathit{elem}\\
&\mathit{nil} = \mathit{Leaf}\\
&\mathit{cons} : \mathit{elem} \to \mathit{Array}\ n\ \mathit{elem} \to \mathit{Array}\ (\mathit{succ}\ n)\ \mathit{elem}\\
&\mathit{cons}\ x_0\ (\mathit{Leaf}) = \mathit{Node}_1\ x_0\ \mathit{Leaf}\ \mathit{Leaf}\\
&\mathit{cons}\ x_0\ (\mathit{Node}_1\ x_1\ l\ r) = \mathit{Node}_2\ x_0\ (\mathit{cons}\ x_1\ r)\ l\\
&\mathit{cons}\ x_0\ (\mathit{Node}_2\ x_1\ l\ r) = \mathit{Node}_1\ x_0\ (\mathit{cons}\ x_1\ r)\ l\\
&\mathit{head} : \mathit{Array}\ (\mathit{succ}\ n)\ \mathit{elem} \to \mathit{elem}\\
&\mathit{head}\ \{\mathtt{0}b\}\ (\mathit{Node}_1\ x_0\ l\ r) = x_0\\
&\mathit{head}\ \{n\,\mathtt{1}\}\ (\mathit{Node}_2\ x_0\ l\ r) = x_0\\
&\mathit{head}\ \{n\,\mathtt{2}\}\ (\mathit{Node}_1\ x_0\ l\ r) = x_0\\
&\mathit{tail} : \mathit{Array}\ (\mathit{succ}\ n)\ \mathit{elem} \to \mathit{Array}\ n\ \mathit{elem}\\
&\mathit{tail}\ \{\mathtt{0}b\}\ (\mathit{Node}_1\ x_0\ l\ r) = \mathit{Leaf}\\
&\mathit{tail}\ \{n\,\mathtt{1}\}\ (\mathit{Node}_2\ x_0\ l\ r) = \mathit{Node}_1\ (\mathit{head}\ l)\ r\ (\mathit{tail}\ l)\\
&\mathit{tail}\ \{n\,\mathtt{2}\}\ (\mathit{Node}_1\ x_0\ l\ r) = \mathit{Node}_2\ (\mathit{head}\ l)\ r\ (\mathit{tail}\ l)
\end{aligned}
$$

Consider the definition of *cons*. Both recursive calls of *cons* are applied to the right sub-tree, additionally swapping left and right subtrees. Of course! Adding an element to the front requires re-indexing: elements that were at even positions before are now located at odd positions, and vice versa. Figure 2 shows *cons* in action, replicating Figure 1 for Braun trees. Observe how the lowest level is gradually populated with elements. Can you identify a pattern?

Both one-two trees and Braun trees are based on the 1-2 number system. To illustrate how intimately the two datastructures are related, consider the effect of two consecutive *cons* operations:

one-two trees:

$$
\begin{aligned}
&\mathit{cons}\ a\ (\mathit{cons}\ b\ (\mathit{Node}_1\ c\ l\ r))\\
\equiv\ &\{\ \text{definition of } \mathit{cons}\ \}\\
&\mathit{cons}\ a\ (\mathit{Node}_2\ b\ c\ l\ r)\\
\equiv\ &\{\ \text{definition of } \mathit{cons}\ \}\\
&\mathit{Node}_1\ a\ (\mathit{cons}\ b\ l)\ (\mathit{cons}\ c\ r)
\end{aligned}
$$

Braun trees:

$$
\begin{aligned}
&\mathit{cons}\ a\ (\mathit{cons}\ b\ (\mathit{Node}_1\ c\ l\ r))\\
\equiv\ &\{\ \text{definition of } \mathit{cons}\ \}\\
&\mathit{cons}\ a\ (\mathit{Node}_2\ b\ (\mathit{cons}\ c\ r)\ l)\\
\equiv\ &\{\ \text{definition of } \mathit{cons}\ \}\\
&\mathit{Node}_1\ a\ (\mathit{cons}\ b\ l)\ (\mathit{cons}\ c\ r)
\end{aligned}
$$

.

One-two trees may be characterized as *lazy Braun trees*: the first *cons* operation is in a sense delayed with the data stored in a two-node. The next *cons* forces the delayed call, issuing two recursive calls. By contrast, *cons* for Braun trees recurses immediately, but makes only a single call. However, after two steps—swapping the sub-trees twice—the net effect is the same. The strategy, lazy or eager, determines the performance: *cons* for Braun trees has a *worst-case* running time of $\Theta(\log n)$, whereas *cons* for one-two trees achieves the logarithmic time bound only in an *amortized* sense.

In this paper, we focus on one-sided flexible arrays. Braun trees are actually more flexible (pun intended) as they also support extension at the rear—they

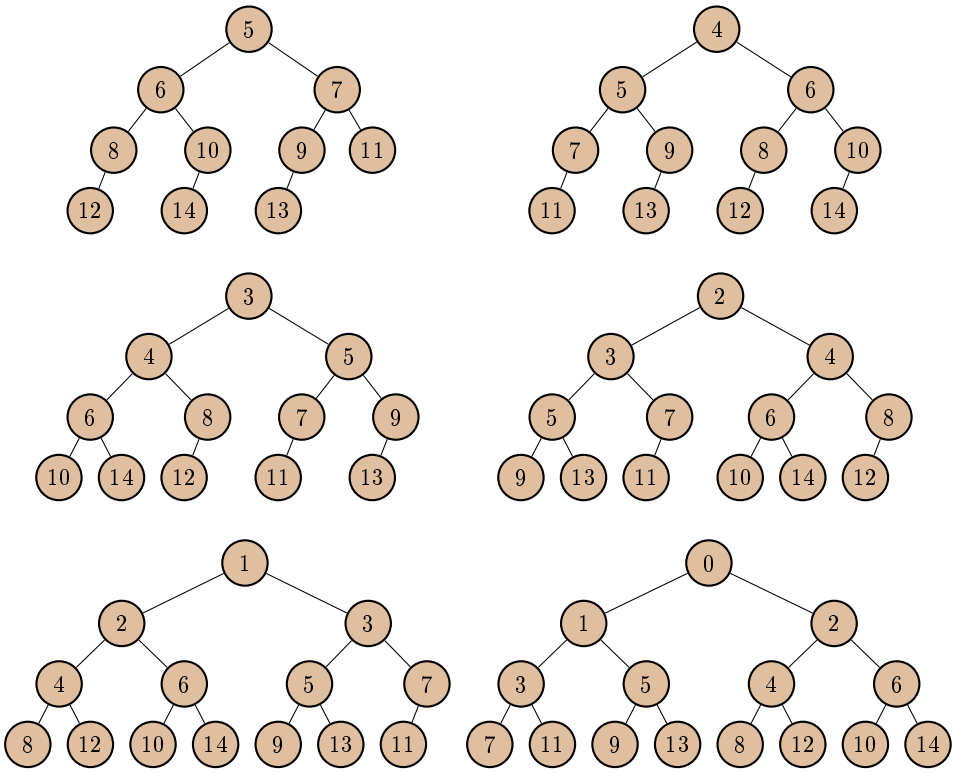
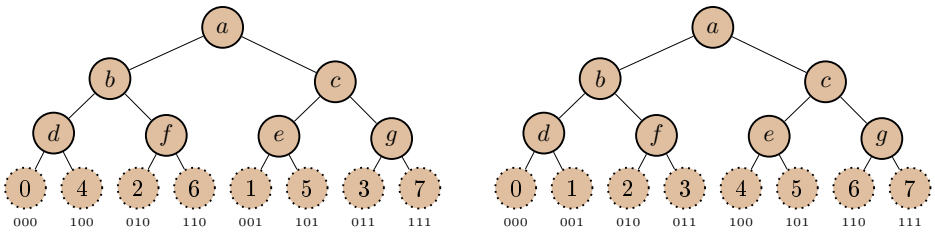


Fig. 2. Braun trees of shape $0b122$ up to $0b1111$.

implement *two-sided flexible arrays*. Through the lens of the interface, *cons* and *snoc*[7] are perfectly symmetric. However, due to the layout of the data, their implementation for Braun trees is quite different. If an element is attached to the front, the positions, odd or even, of the original elements change: sub-trees must be swapped. By contrast, if an element is added to the rear, nothing changes: the sub-trees must stay in place. Depending on the size of the array, the position of the new element is either located in the left or in the right sub-tree, see Figure 2. Staring at the sequence of trees, the position of the last element, the number 14, may appear slightly chaotic. Perhaps this is worth a closer look. Consider the diagram on the left below. The numbers indicate the order in which the positions on the lowest level are filled.



By contrast, the diagram on the right displays the "standard", left-to-right ordering. Comparing the diagrams, we observe that the positions of corresponding nodes are bit-reversals of each other, for example, position $6 = (110)_2$ on the left corresponds to $3 = (011)_2$ on the right. The reason is probably clear by now: the layout of Braun trees is based on zipping (indexing LSB first), whereas the "standard" layout is based on appending (indexing MSB first).

Turning to the implementation of *snoc*, the code is pretty straightforward since the data constructors carry the required size information: if the original size is odd, the element is added to the left sub-tree, if the size is even, it is added to the right sub-tree.

$$
\begin{aligned}
&\mathit{snoc} : \mathit{Array}\ n\ \mathit{elem} \to \mathit{elem} \to \mathit{Array}\ (\mathit{succ}\ n)\ \mathit{elem} \\
&\mathit{snoc}\ (\mathit{Leaf}) \qquad\quad x_n &&= \mathit{Node}_1\ x_n\ \mathit{Leaf}\ \mathit{Leaf} \\
&\mathit{snoc}\ (\mathit{Node}_1\ x_0\ l\ r)\ x_n &&= \mathit{Node}_2\ x_0\ (\mathit{snoc}\ l\ x_n)\ r \\
&\mathit{snoc}\ (\mathit{Node}_2\ x_0\ l\ r)\ x_n &&= \mathit{Node}_1\ x_0\ l\ (\mathit{snoc}\ r\ x_n)
\end{aligned}
$$

As the relative order of x_0, l, r, and x_n must not be changed, each equation is actually forced upon us! (The power of dependent types is particularly tangible if the code is developed interactively.)

The implementation of *snoc* shows that the cases for Node_1 and Node_2 are not necessarily equal! This has consequences when porting the code to non-dependently typed languages such as Haskell—depending on the interface explicit size information may or may not be necessary.

[7] It is customary to call the extension at the rear *snoc*, which is *cons* written backwards.

Remark 6 (Haskell). Continuing the discussion of Remark 5, let us again translate Agda into Haskell code. The implementation of *snoc* has demonstrated that we cannot simply identify $Node_1$ and $Node_2$. For a Haskell implementation there are at least three options:

- we identify the constructors $Node_1$ and $Node_2$ but maintain explicit size information, either locally in each node or globally for the entire tree; or
- we identify the two constructors and recreate the size information on the fly—this can be done in $\Theta(\log^2 n)$ time [20]; or
- we faithfully copy the Agda code at the cost of some code duplication. The duplication of code can, however, be ameliorated using or-patterns.[8]

If we rather arbitrarily select the second option,

```
data Array :: Type → Type where
  Leaf  :: Array elem
  Node  :: elem → Array elem → Array elem → Array elem
```

the implementation of *lookup* is short and sweet,

```
lookup :: Array elem → (Integer → elem)
lookup (Node x0 l r) i | i ≡ 0             = x0
                       | i `mod` 2 ≡ 1 = lookup l ((i - 1) `div` 2)
                       | i `mod` 2 ≡ 0 = lookup r ((i - 2) `div` 2)
```

whereas the definition of *snoc* is more involved.

```
snoc :: Array a → a → Array a
snoc xs xn = put xs (size xs)
  where put (Leaf)       n     = Node xn Leaf Leaf
        put (Node a l r) n
          | n `mod` 2 ≡ 1 = Node a (put l ((n - 1) `div` 2)) r
          | n `mod` 2 ≡ 0 = Node a l (put r ((n - 2) `div` 2))
```

Unfortunately, the running time of *snoc* degrades to $\Theta(\log^2 n)$, as it is dominated by the initial call to *size*. □

10 Random-Access Lists

Both one-two trees and Braun trees are based on the binary `1-2` number system: the types are tries for (two different) index sets; the operations are based on the arithmetic operations. Alas, as already noted, the operations on sequences do not quite achieve the efficiency of their arithmetic counterparts. While the binary increment runs in constant time (in an amortized sense), consing takes logarithmic time (amortized for one-two trees and worst-case for Braun trees).

[8] Unfortunately, Haskell does not support or-patterns but they can be simulated using view patterns and pattern synonyms.

The culprit is easy to identify: the *cons* operation makes two recursive calls for each carry (eagerly or lazily), whereas *incr* makes do with only one. There are two recursive calls as we introduced two recursive sub-trees when we invoked (an instance of) the sum law during triefication, see Section 8:

$$\mathit{law\text{-}of\text{-}exponents\text{-}}\uplus \;:\; (A \times 2 \to X) \cong (A \to X)^{\,2}$$

where $A \times 2 = A \uplus A$ and $A^{\,2} = A \times A$. The isomorphism states that a finite map whose domain has an even size can be represented by two maps whose domains have half the size. If you know the laws of exponents by heart, then you may realize that this is not the only option. Alternatively, we could replace the finite map by a single map that yields pairs. The formal property is a combination of the product rule also known as currying, *law-of-exponents-*$\times$, and the sum rule:

$$\mathit{law\text{-}of\text{-}exponents\text{-}sq} \;:\; (A \times 2 \to X) \cong (A \to X^{\,2})$$

Building on this isomorphism the trieification of the *original* index set of Section 6.2 proceeds as follows.

$$\begin{array}{l}
\mathit{trieify}\ \mathit{elem}\ (n\,\mathtt{1}) \;=\; \\
\quad \textbf{\textit{proof}} \\
\qquad (\mathit{Index}\ (n\,\mathtt{1}) \;\to\; \mathit{elem}) \\
\quad \cong\langle\ \mathit{Index}\text{-}\mathtt{1} \cong\to\cong \cong\text{-}\mathit{reflexive}\ \rangle \\
\qquad (\top \uplus (\mathit{Index}\ n \times 2) \;\to\; \mathit{elem}) \\
\quad \cong\langle\ \mathit{law\text{-}of\text{-}exponents\text{-}}\uplus\ \rangle \\
\qquad (\top \to \mathit{elem}) \times (\mathit{Index}\ n \times 2 \to \mathit{elem}) \\
\quad \cong\langle\ \mathit{law\text{-}of\text{-}exponents\text{-}}\top \cong\times\cong \mathit{law\text{-}of\text{-}exponents\text{-}sq}\ \rangle \\
\qquad \mathit{elem} \times (\mathit{Index}\ n \;\to\; \mathit{elem}^{\,2}) \\
\quad \cong\langle\ \cong\text{-}\mathit{reflexive} \cong\times\cong \mathit{trieify}\ (\mathit{elem}^{\,2})\ n\ \rangle \\
\qquad \mathit{elem} \times \mathit{Array}\ n\ (\mathit{elem}^{\,2}) \\
\quad \cong\langle\ \mathit{use\text{-}as\text{-}definition\text{-}of}\ \mathit{Array}\text{-}\mathtt{1}\ \rangle \\
\qquad \mathit{Array}\ (n\,\mathtt{1})\ \mathit{elem} \\
\quad \blacksquare
\end{array}$$

A similar calculation yields $\mathit{Array}\ (n\,\mathtt{2})\ \mathit{elem} \cong \mathit{elem} \times \mathit{elem} \times \mathit{Array}\ n\ (\mathit{elem}^{\,2})$ for the second inductive case.

All in all, we obtain the following datatype, which is known as the type of *binary, random-access lists*.

$$\begin{array}{l}
\textbf{\textit{data}}\ \mathit{Array} \;:\; \mathit{Leibniz} \to \mathit{Set} \to \mathit{Set}\ \textbf{\textit{where}} \\
\quad \mathit{Nil} \;\;:\; \mathit{Array}\ \mathtt{0}\,b\ \mathit{elem} \\
\quad \mathit{One} \;:\; \mathit{elem} \qquad\qquad \to\ \mathit{Array}\ n\ (\mathit{elem} \times \mathit{elem}) \;\to\; \mathit{Array}\ (n\,\mathtt{1})\ \mathit{elem} \\
\quad \mathit{Two} \;:\; \mathit{elem} \;\to\; \mathit{elem} \;\to\; \mathit{Array}\ n\ (\mathit{elem} \times \mathit{elem}) \;\to\; \mathit{Array}\ (n\,\mathtt{2})\ \mathit{elem}
\end{array}$$

If we wish to emphasize that our new array type is modelled after the `1-2` system, we might prefer the following equivalent definition:

$$\begin{array}{l}
\textbf{\textit{data}}\ \mathit{Array}' \;:\; \mathit{Leibniz} \to \mathit{Set} \to \mathit{Set}\ \textbf{\textit{where}} \\
\quad \mathit{Nil} \;\;:\; \mathit{Array}'\ \mathtt{0}\,b\ \mathit{elem}
\end{array}$$

$$\begin{aligned}
&\mathit{One} : \mathit{elem}^{\,1} \to \mathit{Array}'\; n\,(\mathit{elem}^{\,2}) \to \mathit{Array}'\;(n\,1)\;\mathit{elem}\\
&\mathit{Two} : \mathit{elem}^{\,2} \to \mathit{Array}'\; n\,(\mathit{elem}^{\,2}) \to \mathit{Array}'\;(n\,2)\;\mathit{elem}
\end{aligned}$$

where $A^{\,1} = A$ and $A^{\,2} = A \times A$.

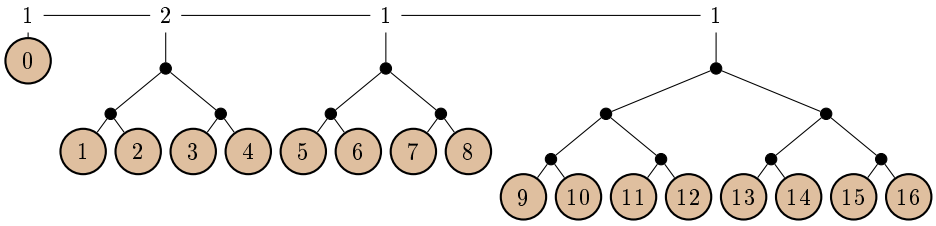
Two remarks are in order. First, our binary numbers are written with the most significant digit on the left. For the array types above, we have reversed the order of bits, as this corresponds to the predominant, left-to-right reading order. Second, our final implementation of arrays is a so-called *nested datatype* [3], where the element type changes at each level. Indeed, a random-access list can be seen as a standard list, except that it contains an element, a pair of elements, a pair of pairs of elements, and so forth. Nested datatypes are also known as non-uniform datatypes [16] or non-regular datatypes [22].

The rest is probably routine by now. As usual, we extract the witnesses of the *trieify* isomorphism, *lookup* and *tabulate*.

$$\begin{aligned}
&\mathit{lookup} : \mathit{Array}\; n\; \mathit{elem} \to (\mathit{Index}\; n \to \mathit{elem})\\
&\mathit{lookup}\,(\mathit{One}\; x_0 \quad\; \mathit{xs})\,(\mathtt{0}\,b_1) = x_0\\
&\mathit{lookup}\,(\mathit{One}\; x_0 \quad\; \mathit{xs})\,(i\,\mathtt{1}_1) = \mathit{proj}_1\,(\mathit{lookup}\; \mathit{xs}\; i)\\
&\mathit{lookup}\,(\mathit{One}\; x_0 \quad\; \mathit{xs})\,(i\,\mathtt{2}_1) = \mathit{proj}_2\,(\mathit{lookup}\; \mathit{xs}\; i)\\
&\mathit{lookup}\,(\mathit{Two}\; x_0\; x_1\; \mathit{xs})\,(\mathtt{0}\,b_2) = x_0\\
&\mathit{lookup}\,(\mathit{Two}\; x_0\; x_1\; \mathit{xs})\,(\mathtt{1}\,b_2) = x_1\\
&\mathit{lookup}\,(\mathit{Two}\; x_0\; x_1\; \mathit{xs})\,(i\,\mathtt{2}_2) = \mathit{proj}_1\,(\mathit{lookup}\; \mathit{xs}\; i)\\
&\mathit{lookup}\,(\mathit{Two}\; x_0\; x_1\; \mathit{xs})\,(i\,\mathtt{3}_2) = \mathit{proj}_2\,(\mathit{lookup}\; \mathit{xs}\; i)
\end{aligned}$$

$$\begin{aligned}
&\mathit{tabulate} : (\mathit{Index}\; n \to \mathit{elem}) \to \mathit{Array}\; n\; \mathit{elem}\\
&\mathit{tabulate}\,\{\mathtt{0}b\}\; f = \mathit{Nil}\\
&\mathit{tabulate}\,\{n\,\mathtt{1}\}\,f = \mathit{One}\,(f\,\mathtt{0}b_1) \qquad (\mathit{tabulate}\,(\lambda\, i \to f(i\,\mathtt{1}_1)\,, f(i\,\mathtt{2}_1)))\\
&\mathit{tabulate}\,\{n\,\mathtt{2}\}\,f = \mathit{Two}\,(f\,\mathtt{0}b_2)\,(f\,\mathtt{1}b_2)\,(\mathit{tabulate}\,(\lambda\, i \to f(i\,\mathtt{2}_2)\,, f(i\,\mathtt{3}_2)))
\end{aligned}$$

The call $\mathit{tabulate}\,\{\mathtt{0}b\;\mathtt{1}\;\mathtt{1}\;\mathtt{2}\;\mathtt{1}\}\;\mathit{id}$ yields the random-access list shown below.



Now the elements appear sequentially from left to right. But wait! Isn't our indexing scheme based on interleaving rather than appending as set out in Section 6.2? This is probably worth a closer look. Let us define a variant of *lookup* that takes its two arguments in reverse order and works on the primed variants of our arrays, defined on the previous page.

$$\begin{aligned}
&\mathit{access} : \mathit{Index}\; n \to \mathit{Array}'\; n\; \mathit{elem} \to \mathit{elem}\\
&\mathit{access}\; i\; t = \mathit{lookup}\; t\; i
\end{aligned}$$

If we compare the implementation of *access* for one-two trees

$$access\,(i\,\mathtt{1}_1)\,(Node_1\,x_0\,xs)\;=\;access\,i\,(proj_1\,xs)$$

to the one for random-access lists,

$$access\,(i\,\mathtt{1}_1)\,(One\,x_0\,xs)\;=\;proj_1\,(access\,i\,xs)$$

we make an interesting observation. The two projection functions are composed in a different order: $access\,i \cdot proj_1$ versus $proj_1 \cdot access\,i$. Of course! This reflects the change in the organisation of data: we have replaced a pair of sub-trees by a sub-tree of pairs. In more detail, the k-th tree of a random-access list corresponds to the k-th level of a one-two tree. As the access order is reversed, the corresponding sequences are bit-reversal permutations of each other. Consider, for example, the lowest level: 9 13 11 15 10 14 12 16 (one-two tree) is the bit-reversal permutation of 9 10 11 12 13 14 15 16 (random-access list).

It is time to reap the harvest. Since our new datastructure is list-like, *cons* makes do with one recursive call.

$$\begin{aligned}
&cons\,:\,elem\,\to\,Array\,n\,elem\,\to\,Array\,(succ\,n)\,elem\\
&cons\,x_0\,(Nil)\;=\;One\,x_0\,Nil\\
&cons\,x_0\,(One\,x_1\;\;xs)\;=\;Two\,x_0\,x_1\,xs\\
&cons\,x_0\,(Two\,x_1\,x_2\,xs)\;=\;One\,x_0\,(cons\,(x_1\,,x_2)\,xs)
\end{aligned}$$

The implementation is truly modelled after the binary increment. This entails, in particular, that *cons* runs in constant amortized time. Figure 3 shows *cons* in action, mirroring Figures 1 and 2. The drawings nicely reflect that a `2` of weight 2^k is equivalent to a `1` of weight 2^{k+1}, see first and third diagram.

If we flip the equations for *cons*, we obtain implementations of *head* and *tail*.

$$\begin{aligned}
&head\,:\,Array\,(succ\,n)\,elem\,\to\,elem\\
&head\,\{\mathtt{0}b\}\;\;(One\,x_0\;\;\;xs)\;=\;x_0\\
&head\,\{n\,\mathtt{1}\}\,(Two\,x_0\,x_1\,xs)\;=\;x_0\\
&head\,\{n\,\mathtt{2}\}\,(One\,x_0\;\;\;xs)\;=\;x_0\\
&tail\,:\,Array\,(succ\,n)\,elem\,\to\,Array\,n\,elem\\
&tail\,\{\mathtt{0}b\}\;\;(One\,x_0\;\;\;xs)\;=\;Nil\\
&tail\,\{n\,\mathtt{1}\}\,(Two\,x_0\,x_1\,xs)\;=\;One\,x_1\,xs\\
&tail\,\{n\,\mathtt{2}\}\,(One\,x_0\;\;\;xs)\;=\;Two\,(proj_1\,(head\,xs))\,(proj_2\,(head\,xs))\,(tail\,xs)
\end{aligned}$$

Observe that we need to make the implicit size arguments explicit, so that Agda is able to distinguish between a singleton array, first equation, and an array that contains at least three elements, third equation. We leave the definition of a suitable list view as the obligatory exercise to the reader (the solution can be found in the accompanying material).

Remark 7 (Haskell). Translating the Agda code to Haskell poses little problems as Haskell supports nested datatypes,

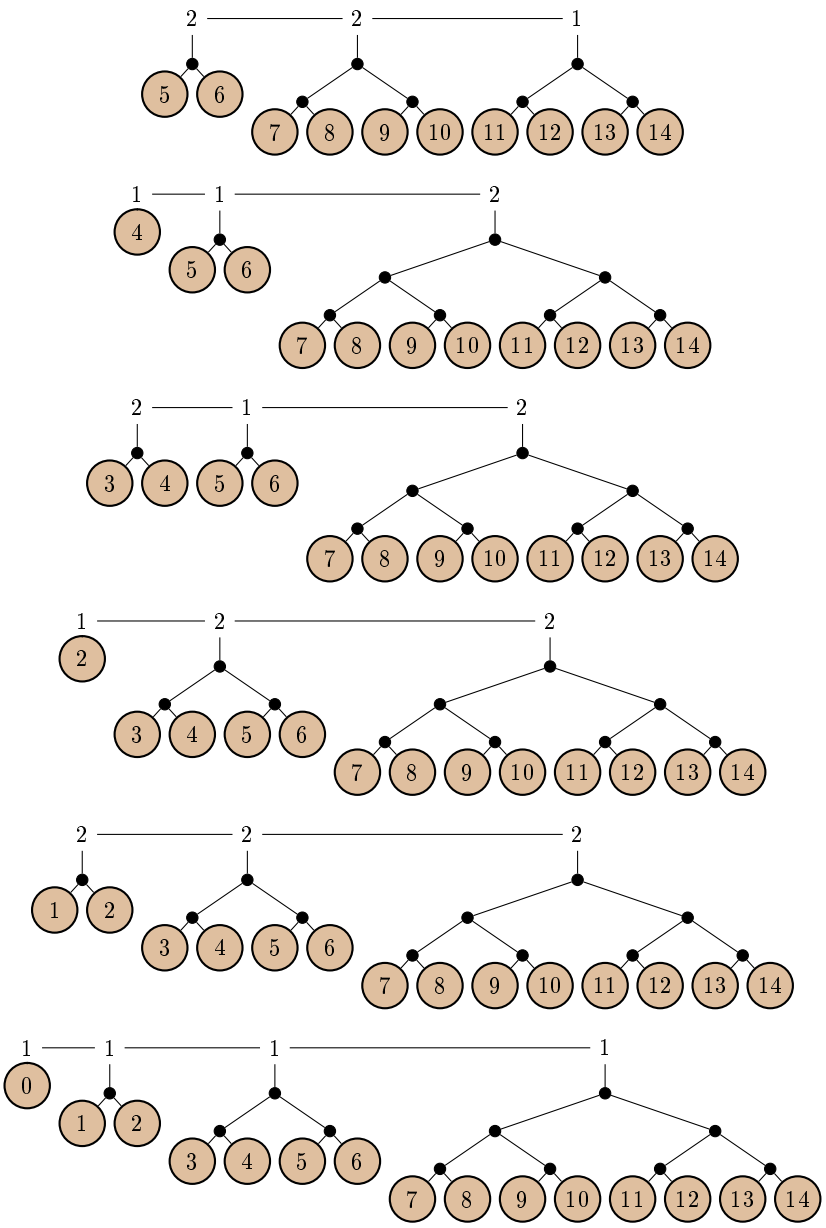


Fig. 3. Random-access lists of shape $0b\,1\,2\,2$ up to $0b\,1\,1\,1\,1$.

$$
\begin{array}{l}
\textbf{\textit{data}}\ \mathit{Array} :: \mathit{Type} \to \mathit{Type}\ \textbf{\textit{where}} \\
\quad \mathit{Nil} \ :: \mathit{Array}\ \mathit{elem} \\
\quad \mathit{One} \ :: \mathit{elem} \to \qquad\quad \mathit{Array}\,(\mathit{elem},\ \mathit{elem}) \to \mathit{Array}\ \mathit{elem} \\
\quad \mathit{Two} :: \mathit{elem} \to \mathit{elem} \to \mathit{Array}\,(\mathit{elem},\ \mathit{elem}) \to \mathit{Array}\ \mathit{elem}
\end{array}
$$

and the definition of recursive functions over them.

$$
\begin{array}{lll}
\mathit{lookup} :: \mathit{Array}\ \mathit{elem} \to (\mathit{Integer} \to \mathit{elem}) & & \\
\mathit{lookup}\,(\mathit{One}\ x_0 \quad\ \mathit{xs})\ i & \mid i \equiv 0 & = x_0 \\
& \mid i\ \text{`}\mathit{mod}\text{`}\ 2 \equiv 1 & = \mathit{fst}\ \ (\mathit{lookup}\ \mathit{xs}\ ((i\text{-}1)\ \ \text{`}\mathit{div}\text{`}\ 2)) \\
& \mid i\ \text{`}\mathit{mod}\text{`}\ 2 \equiv 0 & = \mathit{snd}\,(\mathit{lookup}\ \mathit{xs}\ ((i\text{-}2)\ \ \text{`}\mathit{div}\text{`}\ 2)) \\
\mathit{lookup}\,(\mathit{Two}\ x_0\ x_1\ \mathit{xs})\ i & \mid i \equiv 0 & = x_0 \\
& \mid i \equiv 1 & = x_1 \\
& \mid i\ \text{`}\mathit{mod}\text{`}\ 2 \equiv 0 & = \mathit{fst}\ \ (\mathit{lookup}\ \mathit{xs}\ ((i\text{-}2)\ \ \text{`}\mathit{div}\text{`}\ 2)) \\
& \mid i\ \text{`}\mathit{mod}\text{`}\ 2 \equiv 1 & = \mathit{snd}\,(\mathit{lookup}\ \mathit{xs}\ ((i\text{-}3)\ \ \text{`}\mathit{div}\text{`}\ 2))
\end{array}
$$

Note that the definition is not typable in a standard Hindley-Milner system as the recursive call has type $\mathit{Array}\,(\mathit{elem},\ \mathit{elem}) \to (\mathit{Integer} \to (\mathit{elem},\ \mathit{elem}))$, which is a substitution instance of the declared type. The target language must support *polymorphic recursion* [19]. As typability in this system is undecidable [8], Haskell requires the programmer to provide an explicit type signature. □

Random-access lists outperform one-two trees and Braun trees. But, all that glitters is not gold: unlike their rival implementations, random-access lists do not support a *snoc* operation, extending the end of an array. For this added flexibility, we could "symmetrize" the design. The point of departure is a slightly weird number system that features two least significant digits, one at the front and another one at the rear. Inventing some syntax, $\mathtt{1}\,\langle\,\mathtt{2}\,\langle\,\mathtt{0}\,\rangle\,\mathtt{1}\,\rangle\,\mathtt{1}$, for example, represents $1 + 2 \cdot (2 + 2 \cdot 0 + 1) + 1 = 8$. If we trieify a suitable index type based on this number system, we obtain so-called *finger trees* [14, 4]. But that's a story to be told elsewhere.

11 Related work

We are, of course, not the first to observe the connection between number systems and purely functional datastructures. This observation can be traced as far back as early work by Okasaki [21] and Hinze [9, 11]. Indeed, Okasaki writes "*data structures that can be cast as numerical representations are surprisingly common, but only rarely is the connection to a number system noted explicitly*". This paper tries to provide a framework for making this connection explicit.

Nor are not the first to propose such a framework. McBride's work on *ornaments* describes how to embellish a data type with additional information—and even how to transport functions over one data type to work on its ornamented extension. Typical examples include showing how lists arise from decorating natural numbers with additional information; vectors arise from indexing lists with their length. Ko and Gibbons [15] have shown how these ideas can be applied to

describe how binomial heaps arise as *ornaments* on binary numbers. Similarly, binary random-access lists can be implemented systematically in Agda by indexing with a slight variaton of the binary numbers used in this paper [25]. Instead of using the Leibniz numbers presented here, this construction uses a more traditional 'list of bits' to represent binary numbers. The resulting representation is no longer unique, leading to many different representations of zero—and the empty binary random access list accordingly. Without such unique a representation, the isomorphisms described in this paper do not hold.

The datastructures described in this paper are instances of so-called *Napierian functors* [7], more commonly know as *representable functors*. By design each of our datastructures is isomorphic to a functor of the form $P \to A$, for some (fixed) type of positions P. Indeed, this is the key *lookup-tabulate* isomorphism that we use to calculate the different datastructures throughout this paper. Gibbons's work on Napierian functors was driven by describing APL and enforcing size invariants in multiple dimensions. Although quad trees built from binary numbers are briefly mentioned, the different datastructures that can be calculated using positions built from binary numbers remains largely unexplored.

Nor are we the first to explore type isomorphisms. DiCosmo gives an overview of the field in his survey article [5]; Hinze and James have previously shown how to adapt an equational reasoning style to type isomorphisms, using a few principles from category theory. Recent work on *homotopy type theory* [26], where isomorphic types are guaranteed to be equivalent, might facilitate some of the derivations done in this paper, especially when establishing that operations such as *cons* are respected across isomorphic implementations [17].

There is a great deal of literature on datatype generic tries [10, 12, 13]. These tries exploit the same laws of exponentiation that we have used in this paper. Typically, these tries are used for memoising computations, trading time for space, whereas this paper uses the same laws in a novel context: the derivation of datastructures. These datatype generic tries have appeared in the context of dependent types when recognising languages [1, 6] and memoising computations [24]—but their usage is are novel in this context.

12 Conclusion

This paper has uncovered several well known datastructures in a new way. The key technology—tries, number representations, and type isomorphisms—have been known for decades, yet the connection between isomorphic implementations of datastructures has never been made this explicitly. In doing so, we open the door for further derivations and exploration. The traditional `0-1` number system for binary numbers, for example, will lead to a derivation of zero-one trees, leaf-oriented Braun trees, or `0-1` random-access lists. Providing a *generic* solution, however, where the shift in number representation automatically computes new datastructures, remains a subject for further work. Similarly, we have chosen to restrict ourselves to a single size dimension—an obvious question now

arises how these results may be extended to handle matrices and richer nested datastructures.

Acknowledgements

We would like to thank Markus Heinrich for our discussions in the early stages of this work and his help in formalising several Agda proofs. Clare Martin and Colin Runciman gave invaluable feedback on an early draft.

References

1. Abel, A.: Equational reasoning about formal languages in coalgebraic style (2016), submitted to the CMCS 2016 special issue
2. Altenkirch, T.: Representations of first order function types as terminal coalgebras. In: Typed Lambda Calculi and Applications, TLCA 2001. Lecture Notes in Computer Science, vol. 2044, pp. 62–78. Springer Berlin / Heidelberg (2001)
3. Bird, R., Meertens, L.: Nested datatypes. In: Jeuring, J. (ed.) Fourth International Conference on Mathematics of Program Construction, MPC'98, Marstrand, Sweden. Lecture Notes in Computer Science, vol. 1422, pp. 52–67. Springer Berlin / Heidelberg (June 1998)
4. Claessen, K.: Finger trees explained anew, and slightly simplified (functional pearl). In: Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell. p. 31–38. Haskell 2020, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3406088.3409026, https://doi.org/10.1145/3406088.3409026
5. Di Cosmo, R.: A short survey of isomorphisms of types. Mathematical structures in computer science **15**(5), 825–838 (2005)
6. Elliott, C.: Symbolic and automatic differentiation of languages. Proc. ACM Program. Lang. **5**(ICFP) (aug 2021). https://doi.org/10.1145/3473583, https://doi.org/10.1145/3473583
7. Gibbons, J.: Aplicative programming with naperian functors. In: European Symposium on Programming. pp. 556–583. Springer (2017)
8. Henglein, F.: Type inference with polymorphic recursion. ACM Transactions on Programming Languages and Systems **15**(2), 253–289 (April 1993)
9. Hinze, R.: Functional Pearl: Explaining binomial heaps. Journal of Functional Programming **9**(1), 93–104 (1999). https://doi.org/10.1017/S0956796899003317
10. Hinze, R.: Generalizing generalized tries. Journal of Functional Programming **10**(4), 327–351 (2000). https://doi.org/10.1017/S0956796800003713
11. Hinze, R.: Manufacturing datatypes. Journal of Functional Programming **11**(5), 493–524 (2001). https://doi.org/10.1017/S095679680100404X
12. Hinze, R.: Type fusion. In: Pavlovic, D., Johnson, M. (eds.) Thirteenth International Conference on Algebraic Methodology And Software Technology (AMAST 2010). Lecture Notes in Computer Science, vol. 6486, pp. 92–110. Springer Berlin / Heidelberg (2011). https://doi.org/10.1007/978-3-642-17796-5_6
13. Hinze, R.: Adjoint folds and unfolds—an extended study. Science of Computer Programming **78**(11), 2108–2159 (2013). https://doi.org/10.1016/j.scico.2012.07.011
14. Hinze, R., Paterson, R.: Finger trees: a simple general-purpose data structure. Journal of Functional Programming **16**(2), 197–217 (2006). https://doi.org/10.1017/S0956796805005769

15. Ko, H.s., Gibbons, J.: Programming with ornaments. Journal of Functional Programming **27** (2016). https://doi.org/10.1017/S0956796816000307
16. Kubiak, R., Hughes, J., Launchbury, J.: Implementing projection-based strictness analysis. Tech. rep., Department of Computing Science, University of Glasgow (1991)
17. Licata, D.: Abstract types with isomorphic types (2011), https://homotopytypetheory.org/2012/11/12/abstract-types-with-isomorphic-types/
18. McBride, C., McKinna, J.: The view from the left. Journal of functional programming **14**(1), 69–111 (2004)
19. Mycroft, A.: Polymorphic type schemes and recursive definitions. In: Paul, M., Robinet, B. (eds.) Proceedings of the International Symposium on Programming, 6th Colloquium, Toulouse, France. Lecture Notes in Computer Science, vol. 167, pp. 217–228 (1984)
20. Okasaki, C.: Functional Pearl: Three algorithms on Braun trees. Journal of Functional Programming **7**(6), 661–666 (November 1997)
21. Okasaki, C.: Purely Functional Data Structures. Cambridge University Press (1998)
22. Paterson, R.: Control structures from types (April 1994), ftp://santos.doc.ic.ac.uk/pub/papers/R.Paterson/folds.dvi.gz
23. Pickering, M., Érdi, G., Peyton Jones, S., Eisenberg, R.A.: Pattern synonyms. In: Proceedings of the 9th International Symposium on Haskell. pp. 80–91 (2016)
24. van der Rest, C., Swierstra, W.: A completely unique account of enumeration (2022), under review
25. Swierstra, W.: Heterogeneous binary random-access lists. Journal of Functional Programming **30**, e10 (2020). https://doi.org/10.1017/S0956796820000064
26. Univalent Foundations Program, T.: Homotopy Type Theory: Univalent Foundations of Mathematics. https://homotopytypetheory.org/book, Institute for Advanced Study (2013)
27. Wadler, P.: Views: A way for pattern matching to cohabit with data abstraction. In: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. pp. 307–313 (1987)

A Deriving Operations

The definition of *cons* for one-two trees can be systematically inferred using the abstraction function $[\![\_]\!] = \mathit{lookup}$ that maps one-two trees to finite maps. The types dictate that *cons* applied to a one-node returns a two-node, so we need to solve the equation

$$[\![\ \mathit{cons}\ x_0\ (\mathit{Node}_1\ x_1\ l\ r)\ ]\!] \quad \equiv \quad [\![\ \mathit{Node}_2\ x_0'\ x_1'\ l'\ r'\ ]\!]$$

in the unknowns x_0', x_1', l', and r'. To determine the components of the two-node, we conduct a case analysis on the indices, the arguments of the finite maps. The index $i\,2_2$, for example, determines the third component, the left sub-tree. The derivation works towards a situation where we can apply the specification of *cons* (2b).

```
proof
  [[ cons x0 (Node1 x1 l r) ]] (i 2₂)
≡⟨ by-definition ⟩
  [[ cons x0 (Node1 x1 l r) ]] (isucc (i 1₁))
≡⟨ specification-cons-isucc x1 (Node1 x1 l r) (i 1₁)   (2b) ⟩
  [[ Node1 x1 l r ]] (i 1₁)
≡⟨ by-definition ⟩
  [[ l ]] i
■
```

We may conclude that $l' \equiv l$. The other cases are equally straightforward.

The equation for two-nodes

$$[\![\ \mathit{cons}\ x_0\ (\mathit{Node}_2\ x_1\ x_2\ l\ r)\ ]\!] \quad \equiv \quad [\![\ \mathit{Node}_1\ x_0'\ l'\ r'\ ]\!]$$

is more interesting to solve. Consider the index $i'\,1_1$ that determines the second component, the left sub-tree of the one-node. We need to conduct a further case distinction on i'. Otherwise, Agda is not able to figure out the predecessor of $i'\,1_1$, that the equation $\mathit{isucc}\,i = i'\,1_1$ uniquely determines i for a given i'. For the case analysis we use a combined Peano view, on binary numbers and on binary indices, dealing with the inductive case first.

```
  with view n | iview i'
... | as-succ m | as-isucc i =
  proof
    [[ cons x0 (Node2 x1 x2 l r) ]] ((isucc i) 1₁)
  ≡⟨ by-definition ⟩
    [[ cons x0 (Node2 x1 x2 l r) ]] (isucc (i 2₂))
  ≡⟨ specification-cons-isucc x0 (Node2 x1 x2 l r) (i 2₂)   (2b) ⟩
    [[ Node2 x1 x2 l r ]] (i 2₂)
  ≡⟨ by-definition ⟩
    [[ l ]] i
```

```
  ≡⟨ symmetric (specification-cons-isucc x₁ l i)   (2b) ⟩
    ⟦ cons x₁ l ⟧ (isucc i)
  ∎
```

We conclude that $l' \equiv cons\ x_1\ l$. Again, the calculation is straightforward, except that it is not clear why the first element of l' has to be x_1. The answer is simple, the sub-case *as-isucc* fixes the tail of the sequence, its head is determined by the base case *as-izero*. Actually, there are two base cases, featuring almost identical calculations that only differ in the type arguments. (The "problem" can be traced back to the definition of *izero*, which is defined by case analysis on the size index).

```
... | as-zero     | as-izero =
  proof
    ⟦ cons x₀ (Node₂ x₁ x₂ l r) ⟧ (izero {zero} 1₁)
  ≡⟨ by-definition ⟩
    ⟦ cons x₀ (Node₂ x₁ x₂ l r) ⟧ (isucc {0b 2} (0b₂))
  ≡⟨ specification-cons-isucc x₀ (Node₂ x₁ x₂ l r) 0b₂   (2b) ⟩
    ⟦ Node₂ x₁ x₂ l r ⟧ 0b₂
  ≡⟨ by-definition ⟩
    x₁
  ≡⟨ symmetric (specification-cons-izero x₁ l)   (2a) ⟩
    ⟦ cons x₁ l ⟧ (izero {zero})
  ∎
... | as-succ m | as-izero =
  proof
    ⟦ cons x₀ (Node₂ x₁ x₂ l r) ⟧ (izero {succ m} 1₁)
  ≡⟨ by-definition ⟩
    ⟦ cons x₀ (Node₂ x₁ x₂ l r) ⟧ (isucc {(succ m) 2} (0b₂))
  ≡⟨ specification-cons-isucc x₀ (Node₂ x₁ x₂ l r) 0b₂   (2b) ⟩
    ⟦ Node₂ x₁ x₂ l r ⟧ 0b₂
  ≡⟨ by-definition ⟩
    x₁
  ≡⟨ symmetric (specification-cons-izero x₁ l)   (2a) ⟩
    ⟦ cons x₁ l ⟧ (izero {succ m})
  ∎
```

The derivations confirm that $l' \equiv cons\ x_1\ l$.

On a final note, the steps flagged *by-definition* may be omitted as Agda is able to confirm the equalities automatically. But, of course, the equational proofs are targetted at human readers. An after-the-fact proof that *cons* is correct is actually a three-liner, plus a trivial three-liner for (2a) and a straightforward seven-liner for (2b), but would be wholly unsuitable as a *derivation*.