Random-access lists, from EE to FP

Titouan Quennet and Pierre-Évariste Dagand

Université Paris Cité, CNRS, IRIF

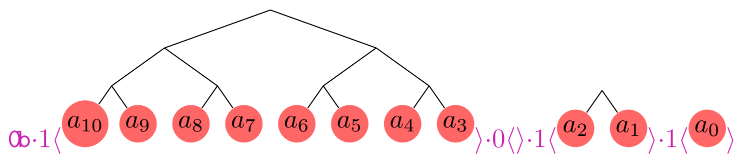
Abstract. What if we could assemble data-types by first picking a recursive structure and then grafting data at selected locations, according to a well-defined blueprint? What if, moreover, this underlying structure had a straightforward semantics in terms of the number of elements it can support? This is the promise of Okasaki's *numerical representations*. This paper offers a journey from Electronic Engineering —computing with binary numbers— to Functional Programming —implementing a persistent random-access list datatype— guided by the type-theoretic framework of McBride's *ornaments*.

Ornaments [Mcbride, 2010] were introduced to dependently-typed programmers as a means to rationalize the development of inductive families. It grew out of the observation that the notion of a "data-type" in a dependently-typed programming language (called an "inductive family" in type theory) can be decomposed into a "data-structure" —a recursive skeleton— onto which is grafted a "data-logic" —witnessing evidences and asserting propositional invariants. This technology exploits a key affordance of modern, dependently-typed programming languages: dependent pattern-matching. Having embedded the logic into the data, one automatically benefits from strong invariants during pattern-matching. The perennial example is the type of vectors, *i.e.* lists indexed by their length. Pattern-matching on a vector reveals not only information about its value but also about its overall length.

The structuring role of ornaments is also at play in proof assistants based on dependent types, such as Coq. Although less critical in this setting, it is frequent for Coq users to define an ML-style inductive type (*e.g.*, the type of lists) onto which an inductive relation is specified (*e.g.*, the predicate `NoDup` in the Coq standard library), so as to *a posteriori* delineate a particular subset of values. This decomposition was, for example, exploited by Dagand et al. [2018] to relate inductive types, inductive predicates, and decision procedures so as to coerce data from one presentation to the other at run-time.

The seminal example of an ornament is the trinity between Peano natural numbers (where natural numbers are inductively presented as either zero or the successor of a natural number), lists (inductively presented as either nil or a cons-cell of some data and another list), and vectors (the inductive family of lists indexed by their length). Interestingly, the relationship between Peano naturals and lists is also studied at the opening of Okasaki [1999] chapter on "Numerical representations" (Chapter 9). Indeed, the gist of numerical representations is to derive data-types (and their operations) out of data-structures encoding a numerical system (and their operations).

For instance, the number "11" is written 0b·1·0·1·1 in binary, with the k^{th} digit being interpreted as a coefficient multiplied by the base 2^k. If we associate, say, a complete leaf binary tree of height h to each digit 1 at position h, we could represent the sequence of 11 elements $[a_0; a_1; a_2; \ldots; a_{10}]$ (in that order) as follows:



(1)

where the digits 1 have been ornamented so as to store a binary tree.

Set in the heart of the nineties, Okasaki's book is not concerned with data-logics: invariants are stated and proved on paper. However, read through the glass of ornaments, numerical representations offer an interesting challenge to our research program: how far reaching is the "data-structure + data-logics" principle in this setting? What are the limits of the exercise, if any?

At the turn of the millennium, Hinze [2001] exploited the expressiveness offered by nested types [Bird and Meertens, 1998] in Haskell to capture the structural invariants of numerical representations [Hinze, 1998]. Some of these techniques have recently been adapted in Agda [Klumpers, 2023] while, independently, Montin et al. [2022] have studied linked nested data-types in Coq.

However, nested types seem to go against the grain of dependent type-theory: working with nested types deprive us from the powerful induction principles offered by inductive families. Defining a recursive program becomes an act of faith where one must constantly struggle to see the tree through a forest of binary products. As a consequence, there has been several efforts to explore numerical representations using inductive families as a more amenable vehicle. An early result is the work of [Ko and Gibbons, 2017], implementing binomial heaps (offering `push` and `pop` operations) from binary numbers with digits 0 and 1. Still in Agda, Swierstra [2020] has implemented random-access lists (offering `cons` and `lookup` operations) from binary numbers with digits 0 and 1.

A sketch of a general framework for calculating data-structures from lookups have been proposed by Hinze and Swierstra [2022] in terms of Napierian logarithms, using binary numbers with digits 1 and 2 as a running example. Similarly, Dagand et al. [2024] sketched a universe of numerical representations, compatible with a type-theoretic presentation but without dwelling on the implementation and specification of supporting operations.

Aside from the aesthetic value of the exercise (which is quite endearing, one must admit!), numerical representations represent an inspiring take on data-type design: we are given a compositional toolbox, by which one chooses a structure (the numerical system), a container of suitable cardinality and, voilà, we ge a set of operations with a strong and clear specification with respect to its cardinality. It is also quite appealing from the point-of-view of property-based testing:

generating an inhabitant of a numerical representation also decomposes into two simpler tasks, first generating a number and, then, generating inhabitants for the data-container.

In the present work, we set our sights onto random-access lists, built upon binary numbers with digits 0 and 1. Unlike previous work, we shall strive to implement all the operations suggested by Okasaki [1999]: cons, hd, tl, lookup, update and drop. This is first and foremost a *pedagogical* exercise for the intellectual delight of our reader, who will enjoy the thrill of getting their neurons for Boolean arithmetic to fire together with the neurons for functional programming. From a practical standpoint, one must caution that, performance-wise, such a representation suffers from uncontrolled ripple carry (discussed further in Section 1.2) while well-established alternatives (such as binary numbers with digits 1 and 2) are widely known [Hinze, 2001, Hinze and Swierstra, 2022].

Another pedagogical choice has been to adopt an idealized notation for an hypothetical programming language based on type theory. The underlying technical development has been carried in the Coq proof assistant[1], following an extrinsic approach (based on inductive relations and their decidability to justify uniqueness of identity proof). However, we felt that the quality of the exposition would have suffered from an unfiltered presentation of the Coq artefact.

First, notationally, indo-arabic numbers are written in a right-to-left manner, *i.e.* with their most-significant digit first and least-significant digit last. This goes against usage in most programming languages, where prefix-based inductive constructors will force a left-to-right style. Second, the objects we study are of interest beyond their current incarnation in Coq: putting proofs aside, functional programmers at large will undoubtedly meet some neat programming puzzles in the following. Finally, an idealized notation allows for some poetic license, which we hope our reader will indulge us with. Following mathematical usage, we have tried to maximize our signal-to-noise ratio by dispensing with unnecessary syntactic details so as to better focus on semantic insights.

Our contributions are the following:

- we give an inductive definition of binary numbers with digits 0 and 1, together with their operations and canonicity properties (Section 1) ;
- we go through the folklore definition of complete leaf binary tree to introduce key notions from the theory of ornaments, focused on structural invariants (Section 2) ;
- we show how binary numbers can be ornamented with binary trees and their operations lifted to recover the usual programming interface of random-access list (Section 4), *i.e.* it cons like a list and support efficient (logarithmic) lookup and update like a random-access structure.

In truth, we simply hope that our readers will be transported, as the binary operations and ourselves did, through the delicate interplay between structure and logical invariants that are at play in the following.

---

[1] Available at `https://github.com/tquennet/random-access-list`. Note to reviewers: in the final version, we will embellish our idealized definitions with hyperlinks to the corresponding Coq definitions typeset with `coqdoc`.

1 Binary numbers

We start our journey with the most bare-bones (and, in fact, cruelly naïve) representation of binary numbers. For now, we make no assumption about the intended semantics of binary numbers: in particular, we shall not attempt to control for trailing zeros at the most-significant position(s).

Being semantics-agnostic allows us to present the type of binary numbers as the combination of a binary choice of digits, either $\mathsf{0}$ or $\mathsf{1}$

$$\begin{aligned}&\mathsf{type}\ \mathsf{Bit} : \star \triangleq\\ &\quad\mid \mathsf{0} : \mathsf{Bit}\\ &\quad\mid \mathsf{1} : \mathsf{Bit}\end{aligned}$$

together with a generic notion of dense representation of numbers, as snoc-lists of digits in a any base D

$$\begin{aligned}&\mathsf{type}\ \mathsf{Num}\ (D : \star) : \star \triangleq\\ &\quad\mid \mathbb{0}_b : \mathsf{Num}\ D\\ &\quad\mid (ds : \mathsf{Num}\ D) \cdot (d : D) : \mathsf{Num}\ D\end{aligned}$$

We thus have binary numbers as a straightforward inductive type

$$\begin{aligned}&\mathsf{Bin} : \star\\ &\mathsf{Bin} \triangleq \mathsf{Num}\ \mathsf{Bit}\end{aligned}$$

For example, this type is inhabited by the likes of "$\mathbb{0}_b$", "$\mathbb{0}_b \cdot 1$", "$\mathbb{0}_b \cdot 1 \cdot 0$", "$\mathbb{0}_b \cdot 1 \cdot 1$" but also, somewhat problematically, "$\mathbb{0}_b \cdot 1 \cdot 0 \cdot 0$", "$\mathbb{0}_b \cdot 0$" and "$\mathbb{0}_b \cdot 0 \cdot 0$".

Expounding the type Num allows us to factor out various notions of iterations over digits through 2 combinators: its functorial action $\mathsf{Num\text{-}mapi}$ and its foldable $\mathsf{Num\text{-}fold}$, whose types are as follows:

$$\begin{aligned}\mathsf{Num\text{-}mapi} &: \{A\ B : \star\}(f : \mathbb{N} \to A \to B)(as : \mathsf{Num}\ A) \to \mathsf{Num}\ B\\ \mathsf{Num\text{-}fold} &: \{M : \star\}\{\mathsf{Monoid}\ M\}(ms : \mathsf{Num}\ M) \to M\end{aligned}$$

Operationally, $\mathsf{Num\text{-}mapi}$ maps a function $f\ k$ for each coefficient k of the number, starting from $k = 0$ for the least-significant digit. $\mathsf{Num\text{-}fold}$ reduces over an arbitrary monoid M. This lets us abstract away the notion of "indexed iteration over all digits" and put forward any non-trivial recursive computations over numbers (for which we will explicitly appeal to recursion). In this paper, our monoid of choice will often be $(\mathbb{N}, +, 0)$, the set of natural numbers with addition and 0. The composition of $\mathsf{Num\text{-}mapi}\ f$ with $\mathsf{Num\text{-}fold}$ is shortened as

$$\mathsf{Num\text{-}foldMap} : \{A\ M : \star\}\{\mathsf{Monoid}\ M\}(f : \mathbb{N} \to A \to M)(as : \mathsf{Num}\ A) \to M$$

If we specialize $\mathsf{Num\text{-}foldMap}$ to the monoid $\star$ (with identity being the unit type and product being the Cartesian product), we obtain the predicate

$$\mathsf{Num\text{-}mapi}^{\square} : \{A : \star\}(P : \mathbb{N} \to A \to \star)(as : \mathsf{Num}\ A) \to \star$$

that asserts that P holds everywhere in as. This corresponds to the "below" predicate transformer [McBride et al., 2004] for the type Num, *i.e.* the predicative counterpart of the functoriality of Num.

The "semantics" of binary numbers is traditionally (*e.g.*, in the Coq standard library [2024b]) given through a recursive function such as

$$
\begin{aligned}
&\mathsf{Bin{\Rightarrow}}\mathbb{N}\ (bs : \mathsf{Bin}) : \mathbb{N}\\
&\mathsf{Bin{\Rightarrow}}\mathbb{N}\ \mathbb{0}\mathsf{b} \qquad\ \triangleq 0\\
&\mathsf{Bin{\Rightarrow}}\mathbb{N}\ (bs{\cdot}\mathsf{0}) \triangleq 2 \times (\mathsf{Bin{\Rightarrow}}\mathbb{N}\ bs)\\
&\mathsf{Bin{\Rightarrow}}\mathbb{N}\ (bs{\cdot}\mathsf{1}) \triangleq 2 \times (\mathsf{Bin{\Rightarrow}}\mathbb{N}\ bs) + 1
\end{aligned}
$$

However, such a definition goes against our ambition to identify and preserve the structure of binary numbers. Here, the weight of the k-th digit of a number is muddled in a stack of k pending multiplications by 2. Besides, we resort to recursion —the GOTO of functional programming— without effecting any computational change to the underlying Num structure. Instead, we prefer the following (equivalent) definition

$$
\begin{aligned}
&\mathsf{Bit{\Rightarrow}}\mathbb{N}\ (k : \mathbb{N})(b : \mathsf{Bit}) : \mathbb{N}\\
&\mathsf{Bit{\Rightarrow}}\mathbb{N}\ k\ \mathsf{0} \triangleq 0 \times 2^k\\
&\mathsf{Bit{\Rightarrow}}\mathbb{N}\ k\ \mathsf{1} \triangleq 1 \times 2^k
\end{aligned}
\qquad
\begin{aligned}
&\mathsf{Bin{\Rightarrow}}\mathbb{N}\ (bs : \mathsf{Bin}) : \mathbb{N}\\
&\mathsf{Bin{\Rightarrow}}\mathbb{N}\ bs \triangleq \mathsf{Num\text{-}foldMap}\ \mathsf{Bit{\Rightarrow}}\mathbb{N}\ bs
\end{aligned}
$$

which will turn into a structural property in Section 4.1.

We recover the intended semantics of binary numbers written right-to-left, following common usage:

$$
\begin{aligned}
\mathsf{Bin{\Rightarrow}}\mathbb{N}\ \mathbb{0}\mathsf{b} &= 0\\
\mathsf{Bin{\Rightarrow}}\mathbb{N}\ (\mathbb{0}\mathsf{b}{\cdot}\mathsf{1}) &= 1\\
\mathsf{Bin{\Rightarrow}}\mathbb{N}\ (\mathbb{0}\mathsf{b}{\cdot}\mathsf{1}{\cdot}\mathsf{0}) &= 2\\
\mathsf{Bin{\Rightarrow}}\mathbb{N}\ (\mathbb{0}\mathsf{b}{\cdot}\mathsf{1}{\cdot}\mathsf{1}) &= 3\\
\mathsf{Bin{\Rightarrow}}\mathbb{N}\ (\mathbb{0}\mathsf{b}{\cdot}\mathsf{1}{\cdot}\mathsf{0}{\cdot}\mathsf{0}) &= 4
\end{aligned}
$$

including warts and all, namely the lack of canonicity of this representation with respect to trailing $\mathsf{0}$s:

$$
\begin{aligned}
\mathsf{Bin{\Rightarrow}}\mathbb{N}\ (\mathbb{0}\mathsf{b}{\cdot}\mathsf{0}) &= 0\\
\mathsf{Bin{\Rightarrow}}\mathbb{N}\ (\mathbb{0}\mathsf{b}{\cdot}\mathsf{0}{\cdot}\mathsf{0}) &= 0\\
\mathsf{Bin{\Rightarrow}}\mathbb{N}\ (\mathbb{0}\mathsf{b}{\cdot}\mathsf{0}{\cdot}\mathsf{1}) &= 1
\end{aligned}
$$

Type theorists, with their compulsive obsession for equality, are quite rightfully keen to run away from such a naïve definition. Instead, they would favor an inductive definition that either enforce the most-significant bit to be $\mathsf{1}$, such as in the Coq standard library [2024a], or they would represent binary numbers with digits "1" and "2", dispensing with the digit "0" altogether, such as in the Agda standard library [2024].

1.1 Canonicity

We are no better than run-of-the-mill type theorists. Following deeply-ingrained Pavlovian conditioning, we do not resist the urge of characterizing the set of canonical representatives of binary numbers. Since we are proceeding after the fact, we resort to a pair of inductive predicates:

$$\boxed{\textsf{is-positive}\ (bs : \textsf{Bin})} \qquad \boxed{\textsf{Bin.is-canonical}\ (bs : \textsf{Bin})}$$

$$\textsf{is-positive}\ (0_b{\cdot}1) \qquad \frac{\textsf{is-positive}\ bs}{\textsf{is-positive}\ (bs{\cdot}1)} \qquad \frac{\textsf{is-positive}\ bs}{\textsf{is-positive}\ (bs{\cdot}0)}$$

$$\textsf{Bin.is-canonical}\ 0_b \qquad \frac{\textsf{is-positive}\ bs}{\textsf{Bin.is-canonical}\ bs}$$

which state that a binary number bs is canonical (Bin.is-canonical bs) if it is either null or stricly positive (is-positive bs). Being positive amounts to having a most-significant bit set to 1.

Cast in an ornamental framework [Mcbride, 2010], these inductive predicates are induced by algebraic ornamentation of the type Bin with the pair of Boolean functions deciding the canonicity and strict positivity of a binary number [Dagand et al., 2018]. We thus automatically obtain the decidability of this predicate and, as a consequence, the unicity of its identity proofs (which, to a programmer, is quite a relief).

Remark that the inhabitants of Bin refined by the predicate Bin.is-canonical are isomorphic to a purely inductive definition, as in the Coq standard library:

```
Inductive positive : Set :=
  | xI : positive -> positive
  | xO : positive -> positive
  | xH : positive.
```

```
Inductive N : Set :=
  | N0 : N
  | Npos : positive -> N.
```

This is indeed a very reasonable implementation of binary numbers. In the present work, we strive to balance clarity of exposition and generality, hinting at the fact that our techniques and results reach beyond binary numbers. This was our motivation for expounding the type Num. The refinement-based approach follows this line of thought: we specialize the generic representation (based on Num) with a domain-specific data-logic, here tailored to binary numbers with digits 0 and 1. While different "terms and conditions" would apply for other numerical systems, the same process of characterizing canonical forms through algebraic ornamentation by a decision procedure is at work.

Given any binary number, we can turn it into an equivalent canonical binary number by trimming away potential trailing $\mathsf{0}$s:

$$\begin{array}{lll}
\mathsf{normalize}\ (bs : \mathsf{Bin}) : \mathsf{Bin} & & \\
\mathsf{normalize}\ \mathsf{0b} & \triangleq \mathsf{0b} & \\
\mathsf{normalize}\ (bs{\cdot}\mathsf{1}) & \triangleq \mathsf{normalize}\ bs{\cdot}\mathsf{1} & \\
\mathsf{normalize}\ (bs{\cdot}\mathsf{0}) & \triangleq \mathsf{0b} & \text{if } \mathsf{normalize}\ bs = \mathsf{0b} \\
\mathsf{normalize}\ (bs{\cdot}\mathsf{0}) & \triangleq \mathsf{normalize}\ bs{\cdot}\mathsf{0} & \text{otherwise}
\end{array}$$

We easily prove that $\mathsf{normalize}$ preserves the semantics ($\mathsf{Bin{\Rightarrow}\mathbb{N}} \circ \mathsf{normalize} = \mathsf{Bin{\Rightarrow}\mathbb{N}}$) and produces canonical numbers (we have $\mathsf{Bin.is\text{-}canonical}\ (\mathsf{normalize}\ bs)$ for any number bs).

1.2 Operations

We can now equip our numerical system with the usual arithmetic operations, starting with increment.

Increment. We joyfully recover the definition we are all familiar with from attending Computer Architecture in our Bachelor years[2]:

$$\begin{array}{ll}
\mathsf{inc}\ (bs : \mathsf{Bin}) : \mathsf{Bin} & \\
\mathsf{inc}\ \mathsf{0b} & \triangleq \mathsf{0b}{\cdot}\mathsf{1} \\
\mathsf{inc}\ (bs{\cdot}\mathsf{0}) & \triangleq bs{\cdot}\mathsf{1} \\
\mathsf{inc}\ (bs{\cdot}\mathsf{1}) & \triangleq (\mathsf{inc}\ bs){\cdot}\mathsf{0}
\end{array}$$

One easily shows that it preserves the canonicity of its argument and rightfully implements the expected semantics ($\mathsf{Bin{\Rightarrow}\mathbb{N}} \circ \mathsf{inc} = (1+) \circ \mathsf{Bin{\Rightarrow}\mathbb{N}}$).

Interestingly, we also have that *canonical* binary numbers are generated by $\mathsf{0b}$ and inc. In type theoretic terms, this means that the type Bin satisfies Peano's induction principle, meaning that we have the property

$$\begin{array}{l}
\forall\, P : \mathsf{Bin} \to \star. \\
\quad P\ \mathsf{0b} \to \\
\quad (\forall\, bs : \mathsf{Bin}.\ \mathsf{Bin.is\text{-}canonical}\ bs \to P\ bs \to P\ (\mathsf{inc}\ bs)) \to \\
\qquad \forall\, bs : \mathsf{Bin}.\ \mathsf{Bin.is\text{-}canonical}\ bs \to P\ bs
\end{array}$$

Decrement. For the implementation of decrement, we have to spook our colleagues from the Electronic Engineering department: for convenience and genericity, we resort to a failure monad $\mathfrak{m} : \star \to \star$ so as to distinguish an integer underflow (namely, upon failing to decrement $\mathsf{0b}$) from normal operation, where

[2] However long ago that was: like cycling, the skills taught in Computer Architecture can never be forgotten.

we may have to propagate a borrow (namely, upon decrementing a number of the form $bs{\cdot}0$):

$$
\begin{aligned}
&\mathsf{dec}\ (bs : \mathsf{Bin}) : \mathfrak{m}\ \mathsf{Bin}\\
&\mathsf{dec}\ \mathsf{0b} \qquad\ \triangleq \mathsf{fail}\ \text{“underflow”}\\
&\mathsf{dec}\ (\mathsf{0b}{\cdot}1) \triangleq \mathsf{return}\ \mathsf{0b}\\
&\mathsf{dec}\ (bs{\cdot}1) \triangleq \mathsf{return}\ (bs{\cdot}0)\\
&\mathsf{dec}\ (bs{\cdot}0) \triangleq \mathsf{let!}\ bs' = \mathsf{dec}\ bs\ \mathsf{in}\ bs'{\cdot}1
\end{aligned}
$$

Explicitly signaling the underflow is key to preserve canonicity: the absorption of the leading 0s is implicitly performed by the monadic composition (bind) of the failure monad, denoted “$\mathsf{let!}\ - = -\ \mathsf{in}\ -$” here.

Our implementation is proved correct in a typical “partial correctness” manner: if dec successfully produces a binary number then that number is canonical. In our effect-generic setting, “partial correctness” amounts to asking for the monad $\mathfrak{m}$ to provide a predicate lifting [Lindley and Stark, 2005, Maillard, 2019]

$$\mathfrak{m}^{\square} : \forall\, P : A \to \star.\ \forall\, ma : \mathfrak{m}\ A.\ \to \star$$

On the Maybe monad, this corresponds exactly to its “below” predicate, which is trivially true upon failure and asserts that P must hold upon success.

Note that, once again, a trained algorithmist would certainly be staggered by the extreme naivety of these binary numbers: inc suffers from logarithmic carry propagation while dec suffers from logarithmic borrow propagation. If both were to enter into a resonant mode (*e.g.*, being used in a non-monotonic counter), the algorithmic complexity would be rather poor. A well-trodden path [Hinze, 2001] consists in adopting a numerical representation with constant-time increment and decrement, such as Myers' skew binary numbers [Myers, 1983]. Whilst being out of the scope of the present work, the same thought process applies there too.

Decidable total order. Over canonical binary numbers, equality is easily decided: by construction, two numbers are equal if and only they have the same canonical representation. To implement comparison, we definitely part ways with the Electronic Engineering department: having the luxury of manipulating words of fixed bit-width, our esteemed colleagues would build their digital comparator cascading from the most significant bit to the least significant bit [Tex, 2003].

In the Mathematically-structured Programming department, we are working with an inductive, least-significant-bit-first presentation of binary numbers. As a consequence, our implementation is continuation-based —so as to reach the most signifiant bits and carry the results back to least significant bits— and it has to do some additional work to identify which word has the shortest bit-width, if any.

Because programming in type-theory is also the art of collecting evidences, we shall strive to make the result of our comparison function as informative as possible: if we have gtb bs bs', then this means that we can split bs into a pair of a shortest canonical prefix $\mathsf{0b}{\cdot}1{\cdot}p$ (least-significant bits) and longest suffix s (most-significant bits, *i.e.*, we have $bs = s{\cdot}1{\cdot}p$) such that there exists a number

$\mathtt{0b}{\cdot}k$ verifying $n+1+\mathtt{0b}{\cdot}k = \mathtt{1}{\cdot}p$. Note that p itself is not binary number, rather it is the *one-hole context* [Huet, 1997, McBride, 2001] of a binary number, *i.e.* a sequence of bits in cons form. Similarly, it is more natural to determine k from least-significant bit to most-significant bit order, so it comes "inside out" as well.

We thus define an operation

$$\mathsf{gtb} : (bs\ bs' : \mathsf{Bin}) \to \mathfrak{m}\ \{s : \mathsf{Bin}; p : \mathsf{List\ Bit}; k : \mathsf{List\ Bit}\}$$

such that the result triple s, p, and k, if it is defined, satisfy the above specification. We remark, in particular, that $1 + \mathtt{0b}{\cdot}k$ corresponds to the result of the subtraction of bs by bs'. From gtb, we thus derive a binary subtraction operation

$$\mathsf{sub} : (bs\ bs' : \mathsf{Bin}) \to \mathfrak{m}\ \mathsf{Bin}$$

The implementation of gtb proceeds by incrementally moving the zipper over bs towards its most significant bits, until exhaustion of the bits of bs' *and* reaching the closest non-0 bit of bs. This separates bs into canonical prefixes and suffices. Besides, we accumulate the bits of the subtraction, propagating borrows to the most-significant bits when necessary.

2 Interlude: complete binary tree

The insight of numerical *representations* consists in ornamenting the data-structure corresponding to a number system (in our case: binary numbers) with a data container of suitable cardinality (in our case: powers-of-2 elements).

Luckily for us, this is an Introduction to Functional Programming classics: binary tree, which we define as follows

$$\begin{array}{l} \mathsf{type\ Tree}\ (A : \star) : \star \triangleq \\ \quad \mid \quad \mathsf{leaf}\ (a : A) : \mathsf{Tree}\ A \\ \quad \mid \mathsf{node}\ (l\ r : \mathsf{Tree}\ A) : \mathsf{Tree}\ A \end{array}$$

In the remainder, we should dispense with the polymorphic quantification over the set A. Our treatment is parameterized over this type.

Computing the cardinality of a binary tree is within the reach of any moderately trained large-language model:

$$\begin{array}{l} \mathsf{Tree.card}\ (t : \mathsf{Tree}\ A) : \mathbb{N} \\ \mathsf{Tree.card}\ (\mathsf{leaf}\ a) \quad \triangleq 1 \\ \mathsf{Tree.card}\ (\mathsf{node}\ l\ r) \triangleq \mathsf{Tree.card}\ l + \mathsf{Tree.card}\ r \end{array}$$

However, our interest in binary trees is narrower than this: we are specifically interested in trees whose cardinality is a power of 2. One way to achieve this is to constrain our binary tree to be complete, *i.e.* the height of the left and right subtrees of every node must be equal:

$$\boxed{\mathsf{Tree.is\text{-}valid}\ (h : \mathbb{N})\ (t : \mathsf{Tree}\ A)}$$

$$\mathsf{Tree.is\text{-}valid}\ 0\ (\mathsf{leaf}\ a) \qquad \frac{\mathsf{Tree.is\text{-}valid}\ h\ l \qquad \mathsf{Tree.is\text{-}valid}\ h\ r}{\mathsf{Tree.is\text{-}valid}\ (h+1)\ (\mathsf{node}\ l\ r)}$$

Doing so, we have that

$$\forall\, h : \mathbb{N}.\ \forall\, t : \mathsf{Tree}\ A.\ \mathsf{Tree.is\text{-}valid}\ h\ t \rightarrow \mathsf{Tree.card}\ t = 2^h$$

Note that one could have chosen other data-structures to this effect, such as binomial trees and pennant trees. For our purposes in the present work (Section 4), complete binary trees are amply sufficient.

Create. Initializing a valid tree of height h from a single element is a straightforward recursive process:

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{Tree.create}\ (a : A)(h : \mathbb{N}) : \mathsf{Tree}\ A} \\
\mathsf{Tree.create}\ a\ 0 & \triangleq \mathsf{leaf}\ a \\
\mathsf{Tree.create}\ a\ (n+1) & \triangleq \mathsf{let!}\ t = \mathsf{Tree.create}\ a\ n\ \mathsf{in}\ \mathsf{node}\ t\ t
\end{array}$$

Lookup. Given a valid binary tree of height h, a list of bits of length h designates a specific element of the tree:

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{Tree.lookup}\ (t : \mathsf{Tree}\ A)(k : \mathsf{List\ Bit}) : \mathfrak{m}\ A} \\
\mathsf{Tree.lookup}\ (\mathsf{leaf}\ a)\ [] & \triangleq \mathsf{return}\ a \\
\mathsf{Tree.lookup}\ (\mathsf{node}\ l\ _)\ (0 :: k) & \triangleq \mathsf{Tree.lookup}\ l\ k \\
\mathsf{Tree.lookup}\ (\mathsf{node}\ _\ r)\ (1 :: k) & \triangleq \mathsf{Tree.lookup}\ r\ k \\
\mathsf{Tree.lookup}\ _\ _ & \triangleq \mathsf{fail}
\end{array}$$

Note that, because we are following an extrinsic approach, we have to prove *a posteriori* that Tree.lookup will in fact always succeeds on a conjunction of a valid tree and an index in the correct range. Similarly, we can update the value stored at a particular index in a tree of suitable height through

$$\mathsf{Tree.update} : (t : \mathsf{Tree}\ A)(k : \mathsf{List\ Bit})(a : A) \rightarrow \mathfrak{m}\ (\mathsf{Tree}\ A)$$

At this stage of the presentation, we thus have a numerical system based on binary numbers (Num) and a data-structure representing collections of powers-of-2 elements (Tree). It is time to put $\mathsf{0b10}$ and 2^h together!

4 Random-access list

The first step consists in extending individual Bits with the data-container Tree. To do so, we ornament the former with a data-extension of the latter:

$$\begin{aligned} &\mathsf{type}\ \mathsf{ArrayBit}\ (A : \star) : \star \triangleq \\ &\quad \mid \qquad\qquad 0\langle\rangle : \mathsf{ArrayBit}\ A \\ &\quad \mid 1\langle(t : \mathsf{Tree}\ A)\rangle : \mathsf{ArrayBit}\ A \end{aligned}$$

which comes with an ornamental (forgetful) map:

$$\begin{aligned} &\mathsf{ArrayBit{\Rightarrow}Bit}\ (mt : \mathsf{ArrayBit}) : \mathsf{Bit} \\ &\mathsf{ArrayBit{\Rightarrow}Bit}\ 0\langle\rangle \ \triangleq 0 \\ &\mathsf{ArrayBit{\Rightarrow}Bit}\ 1\langle t\rangle \triangleq 1 \end{aligned}$$

Note that, much as Bit was equivalent to $\mathbb{B}$, ArrayBit is essentially an option type. Hence the rather unsurprising ornamentation [Dagand, 2017]. For conciseness, we shall exploit the functoriality and foldability of ArrayBit, seen as an option type:

$$\mathsf{ArrayBit.foldMap} : \{A\ M : \star\}\{\mathsf{Monoid}\ M\}(f : A \to M)(mt : \mathsf{ArrayBit}\ A) \to M$$

The ornamentation lifts functorially across the data-type Num, yielding both the type of random-access lists and a forgetful map to the underlying binary number:

$$\begin{aligned} &\mathsf{ArrayList}\ (A : \star) : \star \\ &\mathsf{ArrayList}\ A \triangleq \mathsf{Num}\ (\mathsf{ArrayBit}\ A) \end{aligned}$$

$$\begin{aligned} &\mathsf{ArrayList{\Rightarrow}Bin}\ (as : \mathsf{ArrayList}\ A) : \mathsf{Bin} \\ &\mathsf{ArrayList{\Rightarrow}Bin}\ as \triangleq \mathsf{Num\text{-}mapi}\ (\lambda\ \_.\ \mathsf{ArrayBit{\Rightarrow}Bit})\ as \end{aligned}$$

The number of elements stored in a random-access list corresponds to the sum the elements stored in each individual binary tree:

$$\begin{aligned} &\mathsf{ArrayList.card}\ (as : \mathsf{ArrayList}\ A) : \mathbb{N} \\ &\mathsf{ArrayList.card}\ as \triangleq \mathsf{Num\text{-}foldMap}\ (\lambda\ \_.\ \mathsf{ArrayBit.foldMap}\ \mathsf{Tree.card})\ as \end{aligned}$$

4.1 Validity & canonicity

Obviously, we would expect the cardinality of a random-access list (as computed by ArrayList.card) to correspond to the value of the underlying binary number (as computed by $\mathsf{Bin{\Rightarrow}}\mathbb{N} \circ \mathsf{ArrayList{\Rightarrow}Bin}$). However, this is only true if the height of the underlying binary trees grow accordingly to the power-of-2 coefficients, as specified by $\mathsf{Bit{\Rightarrow}}\mathbb{N}$ and $\mathsf{Bin{\Rightarrow}}\mathbb{N}$.

To enforce this data-logic, we algebraically ornament the type $\mathsf{ArrayBit}$ with the recursive function $\mathsf{Bit}{\Rightarrow}\mathbb{N}$ and functorially lift this predicate over to Num. This produces the following inductive predicates:

$$\boxed{\mathsf{ArrayBit.is\text{-}valid}\ (h : \mathbb{N})\ (mt : \mathsf{ArrayBit})} \qquad \boxed{\mathsf{ArrayList.is\text{-}valid}\ (as : \mathsf{ArrayList})}$$

$$\mathsf{ArrayBit.is\text{-}valid}\ h\ 0\langle\rangle \qquad \frac{\mathsf{Tree.is\text{-}valid}\ h\ t}{\mathsf{ArrayBit.is\text{-}valid}\ h\ 1\langle t\rangle} \qquad \frac{\mathsf{Num\text{-}mapi}^{\square}\ \mathsf{ArrayBit.is\text{-}valid}\ as}{\mathsf{ArrayList.is\text{-}valid}\ as}$$

Under this proviso that a random-access list as satisfies $\mathsf{ArrayList.is\text{-}valid}\ as$, we indeed have that $\mathsf{Bin}{\Rightarrow}\mathbb{N}\ (\mathsf{ArrayList}{\Rightarrow}\mathsf{Bin}\ as) = \mathsf{ArrayList.card}\ as$. However, much like our original definition of Bin, this representation suffers from a lack of canonicity: for instance, there is an infinite number of representations of the empty list. We recover canonicity by simply enforcing canonicity of the underlying numerical structure:

$$\boxed{\mathsf{ArrayList.is\text{-}canonical}\ \ (as : \mathsf{ArrayList})}$$

$$\frac{\mathsf{Bin.is\text{-}canonical}\ (\mathsf{ArrayList}{\Rightarrow}\mathsf{Bin}\ as)}{\mathsf{ArrayList.is\text{-}canonical}\ \ as}$$

The random-access lists we shall consider will have to be both valid and canonical. We package both invariant in an overarching predicate, the composition [Ko and Gibbons, 2017], also-called the pull-back [Dagand, 2017] of both data-logics:

$$\boxed{\mathsf{is\text{-}well\text{-}formed}\ (as : \mathsf{ArrayList})}$$

$$\frac{\mathsf{ArrayList.is\text{-}canonical}\ \ as \qquad \mathsf{ArrayList.is\text{-}valid}\ as}{\mathsf{is\text{-}well\text{-}formed}\ as}$$

4.2 Operations

While the process of turning binary numbers into an operational object required careful thought and semantics considerations (Section 1.2), providing random-access list with operations amounts to, first, finding an operational counterpart in the world of binary numbers and, then, figuring out where the data stored in the binary trees must be transferred to, so as to preserve validity and semantics.

As explained by Okasaki [1999], we expect a rather uneventful journey when it comes to initializing a random-access list of a given size (ArrayList.create), adding an element to a list (cons) or project it out its head and tail (respectively, hd and tl). The crux of the matter concerns the implementation of the logarithmic lookup and update operations (respectively, lookup and update). Luckily, there turns out to be a single abstraction (open) underpinning both abstractions and this abstraction has an interesting counterpart on the numerical side.

Create. Initializing a random-access list of a given cardinality is dual to converting a random-access list back to a binary number with $\mathsf{ArrayList{\Rightarrow}Bin}$: it consists in turning the digit 1 into a constructor $1\langle - \rangle$ with a binary tree of 2^h elements.

$$
\begin{array}{l}
\mathsf{ArrayBit.create}(a : A)(h : \mathbb{N})(b : \mathsf{Bit}) : \mathsf{ArrayBit}\ A \\
\mathsf{ArrayBit.create}\ a\ h\ 0 \triangleq 0\langle\rangle \\
\mathsf{ArrayBit.create}\ a\ h\ 1 \triangleq 1\langle \mathsf{Tree.create}\ a\ h \rangle
\end{array}
$$

$$
\begin{array}{l}
\mathsf{ArrayList.create}(a : A)(bs : \mathsf{Bin}) : \mathsf{ArrayList}\ A \\
\mathsf{ArrayList.create}\ a\ bs \triangleq \mathsf{Num\text{-}mapi}\ (\mathsf{ArrayBit.create}\ a)\ bs
\end{array}
$$

Given a binary number in canonical form, this results in a well-formed random-access list. In particular, the following diagram commutes:

$$
\begin{tikzcd}
\mathsf{Bin} \arrow[r, "\mathsf{ArrayList.create}\ a"] \arrow[dr, equal] & \mathsf{ArrayList}\ A \arrow[d, "\mathsf{ArrayList{\Rightarrow}Bin}"] \\
& \mathsf{Bin}
\end{tikzcd}
$$

Cons. The implementation of inc (Section 1.2, page 7) turns into a blueprint laying out the recursive definition of the insertion of a valid tree t of height h (*i.e.*, satisfying $\mathsf{Tree.is\text{-}valid}\ h\ t$) into a valid random-access list $0b \cdot \ldots \cdot as_1 \cdot as_0$ of the corresponding weights (*i.e.*, satisfying $\mathsf{ArrayBit.is\text{-}valid}\, h\, as_0$, $\mathsf{ArrayBit.is\text{-}valid}\, (h+1)\ as_1$, *etc.*). The only novelty here is having to manipulate trees, with the occasional carry from one digit to a twice bigger node at the next digit:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{cons\text{-}tree}\ (t : \mathsf{Tree}\ A)(as : \mathsf{ArrayList}\ A) : \mathsf{ArrayList}\ A} \\
\mathsf{cons\text{-}tree}\ t\ 0b & \triangleq 0b \cdot 1\langle t \rangle \\
\mathsf{cons\text{-}tree}\ t\ (as \cdot 0\langle\rangle) & \triangleq as \cdot 1\langle t \rangle \\
\mathsf{cons\text{-}tree}\ t\ (as \cdot 1\langle t' \rangle) & \triangleq (\mathsf{cons\text{-}tree}\ (\mathsf{node}\ t'\ t)\ as) \cdot 0\langle\rangle
\end{array}
$$

Note that we are careful to put the cons-ed tree t *to the right* of the tree t', which was already there. The relation between $\mathsf{cons\text{-}tree}$ and inc amounts to the following commuting diagram:

$$
\begin{tikzcd}
\mathsf{ArrayList} \arrow[r, "\mathsf{cons\text{-}tree}\ t"] \arrow[d, "\mathsf{ArrayList{\Rightarrow}Bin}"] & \mathsf{ArrayList} \arrow[d, "\mathsf{ArrayList{\Rightarrow}Bin}"] \\
\mathsf{Bin} \arrow[r, "\mathsf{inc}"] & \mathsf{Bin}
\end{tikzcd}
$$

Functional ornaments [Dagand and McBride, 2014] and their later evolutions [Williams and Rémy, 2018] offered the promise to assist this process of lifting a recursive definition (here, inc) to an ornamented version of its input and output types (here, $\mathsf{cons\text{-}tree}$) while guaranteeing commutativity with respect to forgetful maps (here, $\mathsf{ArrayList{\Rightarrow}Bin}$). For readability, we chose to expound the actual definition rather than appeal to witchcraft (this article having burned through a sufficiently large quantity of octarine ink already).

To simplify the programming interface, we expose the cons operator instead

$$
\begin{aligned}
&\mathsf{cons}\ (a : A)(as : \mathsf{ArrayList}\ A) : \mathsf{ArrayList}\ A \\
&\mathsf{cons}\ a\ as \triangleq \mathsf{cons\text{-}tree}\ (\mathsf{leaf}\ a)\ as
\end{aligned}
$$

whose invariant is more simply stated as: given a well-formed random-access list as (*i.e.*, satisfying $\mathsf{is\text{-}well\text{-}formed}\ as$) and a single element a, $\mathsf{cons}\ a\ as$ produces a well-formed random-access list whose cardinality has been increased by 1.

As for binary numbers, having implemented $\mathsf{0b}$ and cons, we can prove that well-formed random-access list satisfy the usual induction principle for lists, that is we have:

$$
\begin{aligned}
&\forall\ P : \mathsf{ArrayList}\ A \to \star. \\
&\quad P\ \mathsf{0b} \to \\
&\quad (\forall\ a : A.\ \forall\ as : \mathsf{ArrayList}\ A.\ \mathsf{is\text{-}well\text{-}formed}\ as \to P\ as \to P\ (\mathsf{cons}\ a\ as)) \to \\
&\qquad \forall\ as : \mathsf{ArrayList}\ A.\ \mathsf{is\text{-}well\text{-}formed}\ as \to P\ as
\end{aligned}
$$

Uncons. The dual operation consists in removing the head of the random-access list. Once again, we rely on the implementation of dec as a blueprint for the recursive definition:

$$
\begin{aligned}
&\mathsf{uncons}\ (as : \mathsf{ArrayList}\ A) : \mathfrak{m}\ (\mathsf{Tree}\ A \times \mathsf{ArrayList}\ A) \\
&\mathsf{uncons}\ \mathsf{0b} && \triangleq \mathsf{fail}\ \text{“underflow”} \\
&\mathsf{uncons}\ (\mathsf{0b}{\cdot}1\langle t\rangle) && \triangleq \mathsf{return}\ (t\ ,\ \mathsf{0b}) \\
&\mathsf{uncons}\ (as{\cdot}1\langle t\rangle) && \triangleq \mathsf{return}\ (t\ ,\ as{\cdot}0\langle\rangle) \\
&\mathsf{uncons}\ (as{\cdot}0\langle\rangle) && \triangleq \mathsf{let!}\ (\mathsf{node}\ t\ t'\ ,\ as') = \mathsf{uncons}\ as\ \mathsf{in}\ (t'\ ,\ as'{\cdot}1\langle t\rangle)
\end{aligned}
$$

subject to an ornamental invariant relating the cardinality of the input random-access list to the cardinality of the (potential) output random-access list in terms of dec:

$$
\begin{tikzcd}
\mathsf{ArrayList} \arrow[r, "\mathsf{uncons}"] \arrow[d, "\mathsf{ArrayList{\Rightarrow}Bin}"] & \mathfrak{m}\ (\mathsf{Tree}\ A \times \mathsf{ArrayList}\ A) \arrow[d, "\mathfrak{m}^{\to}(\mathsf{ArrayList{\Rightarrow}Bin} \circ \mathsf{proj}_2)"] \\
\mathsf{Bin} \arrow[r, "\mathsf{dec}"] & \mathfrak{m}\ \mathsf{Bin}
\end{tikzcd}
$$

Compared to dec, the only novelty here consists in splitting binary nodes so as to materialize the borrow from a more significant bit to a least significant bit.

Starting from a well-founded random-access list (whose least significant bit has order 0), we are guaranteed that the first component of the pair is a tree t satisfying $\mathsf{Tree.is\text{-}valid}\ 0\ t$, which, by inversion of the validity predicate, means that it is necessarily a $\mathsf{leaf}\ a$. We expose the following, simpler interface instead:

$$
\begin{aligned}
&\mathsf{hd}\ (as : \mathsf{ArrayList}\ A) : \mathfrak{m}\ A \\
&\mathsf{hd}\ as \triangleq \mathsf{let!}\ (\mathsf{leaf}\ a\ ,\ \_) = \mathsf{uncons}\ as\ \mathsf{in}\ a
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{tl}\ (as : \mathsf{ArrayList}\ A) : \mathfrak{m}\ (\mathsf{ArrayList}\ A) \\
&\mathsf{tl}\ as \triangleq \mathsf{let!}\ (\mathsf{leaf}\ \_\ ,\ as') = \mathsf{uncons}\ as\ \mathsf{in}\ as'
\end{aligned}
$$

Open. The insertion and update operations over random-access list have the following types:

$$\mathsf{lookup} : (as : \mathsf{ArrayList}\ A)(bs : \mathsf{Bin}) \to \mathfrak{m}\ A$$
$$\mathsf{update} : (as : \mathsf{ArrayList}\ A)(bs : \mathsf{Bin})(a : A) \to \mathfrak{m}\ (\mathsf{ArrayList}\ A)$$

In both cases, we must first identify to which non-zero digit of the input list as —if it exists— one must look into so as to further navigate into the associated binary tree in order to find either the element to return (in the case of lookup) or to replace it (in the case of update).

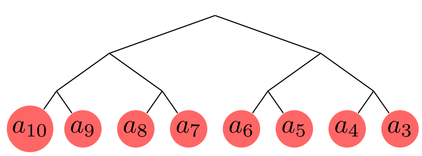
How should we determine whether this position exists? As a matter of fact, this corresponds exactly to the definition of gtb (Section 1.2, page 9), where the first argument is ornamented from a binary number to a random-access list while keeping the second argument unchanged. Ornamenting the result types gives

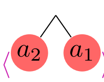
$$\mathsf{open} : (as : \mathsf{ArrayList}\ A)(bs : \mathsf{Bin}) \to \mathfrak{m} \left\{ \begin{array}{l} s : \mathsf{ArrayList}\ A; \\ t : \mathsf{Tree}\ A; \\ p : \mathsf{List}\ (\mathsf{ArrayBit}\ A); \\ k : \mathsf{List}\ \mathsf{Bit} \end{array} \right\}$$

that witnesses the fact that —if bs is smaller than as— the list as can be split into a canonical prefix $1\langle t\rangle{\cdot}p$ (where t is a tree of height h) and a suffix s (*i.e.*, $as = s{\cdot}1\langle t\rangle{\cdot}p$). Besides, $\mathbb{0}\mathsf{b}{\cdot}k$ represents the difference between as and bs: we have $bs + 1 + \mathbb{0}\mathsf{b}{\cdot}k = 1{\cdot}(\mathsf{ArrayList{\Rightarrow}Bin}\ p)$. A direct consequence is that $\mathbb{0}\mathsf{b}{\cdot}k$ denotes a number in the range $\{0, 1, \ldots, 2^h - 1\}$.

Let us illustrate this decomposition on our earlier example of an 11-elements random-access list (Equation 1, p. 2). If we open this list at index 6, we obtain:

$s \triangleq \mathbb{0}\mathsf{b}$

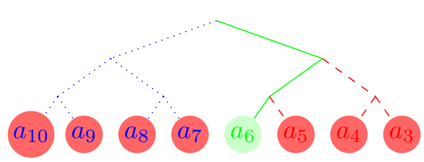
$t \triangleq$ 

$p \triangleq 0\langle\rangle :: 1\langle$ 

$\rangle :: 1\langle a_0 \rangle$

$k \triangleq 1 :: 0 :: 0$

Effectively, k encodes a dissection [McBride, 2008] of the binary tree t: k identifies a specific element of t, where the bit 0 is interpreted as "move to left subtree" and the bit 1 as "move to right subtree". On the left and in blue, are elements which were cons-ed *later* and, on the right and in red, elements which were cons-ed *earlier*:



Note that $\mathbb{0b}{\cdot}k = \mathbb{0b}{\cdot}1{\cdot}0{\cdot}0$ denotes the number 4: indeed, the designated element is the 4^{th} leaf of the tree, counting from right-to-left!

The structural ties between open and gtb is summarized by this commuting diagram:

$$\begin{tikzcd}
\mathsf{ArrayList}\ A \times \mathsf{Bin} \arrow[r, "\mathsf{open}"] \arrow[d, "(\mathsf{ArrayList}{\Rightarrow}\mathsf{Bin},\mathsf{id})"] & \mathfrak{m}\ \{s : \mathsf{ArrayList}\ A; t : \mathsf{Tree}\ A; p : \mathsf{List}\ (\mathsf{ArrayBit}\ A); k : \mathsf{List\ Bit}\} \arrow[d, "\mathfrak{m}^{\rightarrow}\{s:\mathsf{ArrayList}{\Rightarrow}\mathsf{Bin};p:\mathsf{List}^{\rightarrow}\,\mathsf{ArrayBit}{\Rightarrow}\mathsf{Bit};k:\mathsf{id}\}"] \\
\mathsf{Bin} \times \mathsf{Bin} \arrow[r, "\mathsf{gtb}"] & \mathfrak{m}\ \{s : \mathsf{Bin}; p : \mathsf{List\ Bit}; k : \mathsf{List\ Bit}\}
\end{tikzcd}$$

Lookup. Using open, it becomes easier to extract the n^{th} element (in cons-order) of a random-access list as: it is the element designated by k in the binary tree t. To reach this element, we navigate the binary tree according to its path, coded in binary (Section 2, page 10):

$$\begin{array}{l}
\mathsf{lookup}(as : \mathsf{ArrayList}\ A)(bs : \mathsf{Bin}) : \mathfrak{m}\ A \\
\mathsf{lookup}\ as\ bs \triangleq \mathsf{let!}\ \{_\ ;\ t\ ;\ _\ ;\ k\} = \mathsf{open}\ as\ bs\ \mathsf{in} \\
\qquad\qquad\qquad\quad \mathsf{Tree.lookup}\ t\ k
\end{array}$$

Update. The zipper over random-access list really shines in the implementation of update: it allows us to perform the update of the designated tree and then plug this updated tree back into an otherwise unchanged random-access list:

$$\begin{array}{l}
\mathsf{update}(as : \mathsf{ArrayList}\ A)(bs : \mathsf{Bin})(a : A) : \mathfrak{m}\ A \\
\mathsf{update}\ as\ bs\ a \triangleq \mathsf{let!}\ \{s\ ;\ t\ ;\ p\ ;\ k\} = \mathsf{open}\ as\ bs\ \mathsf{in} \\
\qquad\qquad\qquad\quad\ \mathsf{let!}\ t' = \mathsf{Tree.update}\ t\ k\ a\ \mathsf{in} \\
\qquad\qquad\qquad\quad\ \mathsf{return}\ s{\cdot}1\langle t'\rangle{\cdot}p
\end{array}$$

Drop. Okasaki [1999] left as an exercise to the reader the task of implementing drop, which remove the first bs elements of a random-access list as. Once again, we rely on open to deliver a zipper-based decomposition capturing the state "as at index bs". However, whilst the most-significant bits s of as will carry over as-is to the resulting list, we have to rebalance the fragment consisting of the last $k + 1$ elements cons-ed in that tree t (*i.e.*, the k elements to the left of the designated element and the designated element itself) onto a new prefix. The simplest way to achieve this is to first unload the k elements to the left of the designated value (keeping this value aside for the moment) in a the one-hole context of a random-access list. Computing this structure is straightforward: it is coded upon k itself! This corresponds to the following operation:

$$\begin{array}{ll}
\mathsf{Tree.scatter}\ (t : \mathsf{Tree}\ A)(bs : \mathsf{List}\ \mathbb{B}) : \mathfrak{m}\ (A \times \mathsf{List}\ (\mathsf{ArrayBit}\ A)) & \\
\mathsf{Tree.scatter}\ (\mathsf{leaf}\ a)\ [] & \triangleq (a\ ,\ []) \\
\mathsf{Tree.scatter}\ (\mathsf{node}\ lt\ rt)\ (1 :: bs) & \triangleq \mathsf{let!}\ (a\ ,\ p) = \mathsf{Tree.scatter}\ rt\ bs\ \mathsf{in}\ (a\ ,\ 1\langle lt\rangle :: p) \\
\mathsf{Tree.scatter}\ (\mathsf{node}\ lt\ _)\ (0 :: bs) & \triangleq \mathsf{let!}\ (a\ ,\ p) = \mathsf{Tree.scatter}\ lt\ bs\ \mathsf{in}\ (a\ ,\ 0\langle\rangle :: p)
\end{array}$$

which returns the value at the designated element (a) together with the new prefix p of k elements. The random-access list $s \cdot 0\langle\rangle \cdot p$ is almost the desired result but we are short of one element: a. We must cons it to the overall list (which may lead to cascading overflows up to the separating $0\langle\rangle$):

$$
\begin{array}{l}
\mathsf{drop}\ (as : \mathsf{ArrayList}\ A)(bs : \mathsf{Bin}) : \mathfrak{m}\ (\mathsf{ArrayList}\ A) \\
\mathsf{drop}\ as\ bs \triangleq \mathsf{let!}\ \{s\ ;\ t\ ;\ \_\ ;\ k\} = \mathsf{open}\ as\ bs\ \mathsf{in} \\
\qquad\qquad\quad \mathsf{let!}\ (a\ ,\ p') = \mathsf{Tree.scatter}\ t\ k\ \mathsf{in} \\
\qquad\qquad\quad \mathsf{return}\ (\mathsf{cons}\ a\ (s \cdot 0\langle\rangle \cdot p'))
\end{array}
$$

4.3 Equational theory

Proving the equational theory of random-access list is “business as usual”, with the notable exception that 1. we can rely on the ornamental projection to binary number whenever necessary (*e.g.*, to characterize out-of-bound accesses as binary overflow) and 2. the trio of operations lookup, update and drop rely on a single, common abstraction open. Developing the equational theory through open yields more general proofs.

8 Conclusion

This journey from binary numbers to random-access lists raises some interesting questions, which we side-stepped here thanks to carefully-crafted definitions and *a posteriori* proofs that our craft was correct. This suggests that we are relying on some implicit invariants, which we recovered after the fact through sheer obstinacy. An intrinsic presentation would have been much less forgiving. In fact, all our attempts at giving a maximally-ornamented intrinsic presentation of random-access list in Agda have failed so far. A maximally-ornamented intrinsic random-access list is a random-access list indexed by the binary number encoding its size: by construction, any operation on those is ornamentally-correct. However, our experience has been that it can be extremely challenging (quite an understatement!) to work with pattern-matching to get through recursive steps. The work of Hinze and Swierstra [2022] is an obvious inspiration for future work: in their work, the authors have expounded some of the implicit invariants at play between the type of indices and data-types supporting such indices.

Our presentation relied extensively on ornaments as a conceptual apparatus but did not use it as an effective tool. Whenever ornamentation manifests itself, we manually roll our own according to the ornamental blueprints on paper. With proper language support [Dagand and McBride, 2014, Ko and Gibbons, 2017, Williams and Rémy, 2018], we would have avoided this extra legwork. However, to the best of our knowledge, existing ornament library or ornament-processing systems are still to verbose given our pedagogical intents in the present work.

Bibliography

Agda standard library. Binary numbers, 2024. Module `Data.Nat.Binary.Base` (l.29-32).

R. S. Bird and L. G. L. T. Meertens. Nested datatypes. In J. Jeuring, editor, *Mathematics of Program Construction, MPC'98, Marstrand, Sweden, June 15-17, 1998, Proceedings*, volume 1422 of *Lecture Notes in Computer Science*, pages 52–67. Springer, 1998. `https://doi.org/10.1007/BFb0054285`.

Coq standard library. Binary numbers, 2024a. Module `Numbers.BinNums` (l.21-24).

Coq standard library. Binary numbers: `to_nat`, 2024b. Module `NArith.BinNatDef` (l.367-371).

P. Dagand. The essence of ornaments. *J. Funct. Program.*, 27:e9, 2017. `https://doi.org/10.1017/S0956796816000356`.

P. Dagand and C. McBride. Transporting functions across ornaments. *J. Funct. Program.*, 24(2-3):316–383, 2014. `https://doi.org/10.1017/S0956796814000069`.

P. Dagand, N. Tabareau, and É. Tanter. Foundations of dependent interoperability. *J. Funct. Program.*, 28:e9, 2018. `https://doi.org/10.1017/S0956796818000011`.

P.-E. Dagand, P. Letouzey, and E. F. Taghayor. Rough Pearl: Manufacturing Cons-Cells. In *35es Journées Francophones des Langages Applicatifs (JFLA 2024)*, 2024. URL `https://inria.hal.science/hal-04406422`.

R. Hinze. Numerical representations as higher-order nested datatypes. Technical Report IAI-TR-98-12, Institut für Informatik III, Universität Bonn, 1998. URL `https://www.cs.ox.ac.uk/ralf.hinze/publications/IAI-TR-98-12.ps.gz`.

R. Hinze. Manufacturing datatypes. *J. Funct. Program.*, 11(5):493–524, 2001. `https://doi.org/10.1017/S095679680100404X`.

R. Hinze and W. Swierstra. Calculating datastructures. In *Mathematics of Program Construction MPC*, 2022. `https://doi.org/10.1007/978-3-031-16912-0_3`.

G. P. Huet. The zipper. *J. Funct. Program.*, 7(5), 1997. `https://doi.org/10.1017/S0956796897002864`.

S. Klumpers. Generic numerical representations as ornaments. Master's thesis, Utrecht University, 2023. URL `https://studenttheses.uu.nl/handle/20.500.12932/45670`.

H. Ko and J. Gibbons. Programming with ornaments. *J. Funct. Program.*, 27:e2, 2017. `https://doi.org/10.1017/S0956796816000307`.

S. Lindley and I. Stark. Reducibility and tt-lifting for computation types. In P. Urzyczyn, editor, *Typed Lambda Calculi and Applications, 7th International Conference, TLCA 2005, Nara, Japan, April 21-23, 2005, Proceedings*, volume 3461 of *Lecture Notes in Computer Science*, pages 262–277. Springer, 2005. `https://doi.org/10.1007/11417170_20`.

K. Maillard. *Principles of Program Verification for Arbitrary Monadic Effects. (Principes de la Vérification de Programmes à Effets Monadiques Arbitraires)*. PhD thesis, École Normale Supérieure, Paris, France, 2019. URL `https://tel.archives-ouvertes.fr/tel-02416788`.

C. McBride. The Derivative of a Regular Type is its Type of One-Hole Contexts. Available at `http://strictlypositive.org/diff.pdf`, 2001.

C. McBride. Clowns to the left of me, jokers to the right (pearl): dissecting data structures. In G. C. Necula and P. Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 287–295. ACM, 2008. `https://doi.org/10.1145/1328438.1328474`.

C. Mcbride. Ornamental algebras, algebraic ornaments. Manuscript available online, 2010. URL `http://personal.cis.strath.ac.uk/~conor/pub/OAAO/Ornament.pdf`.

C. McBride, H. Goguen, and J. McKinna. A few constructions on constructors. In J. Filliâtre, C. Paulin-Mohring, and B. Werner, editors, *Types for Proofs and Programs, International Workshop, TYPES 2004, Jouy-en-Josas, France, December 15-18, 2004, Revised Selected Papers*, volume 3839 of *Lecture Notes in Computer Science*, pages 186–200. Springer, 2004. `https://doi.org/10.1007/11617990_12`.

M. Montin, A. Ledein, and C. Dubois. Libndt: Towards a formal library on spreadable properties over linked nested datatypes. In J. Gibbons and M. S. New, editors, *Proceedings Ninth Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2022, Munich, Germany, 2nd April 2022*, volume 360 of *EPTCS*, pages 27–44, 2022. `https://doi.org/10.4204/EPTCS.360.2`.

E. W. Myers. An applicative random-access stack. *Inf. Process. Lett.*, 17(5): 241–248, 1983. `https://doi.org/10.1016/0020-0190(83)90106-0`.

C. Okasaki. *Purely functional data structures*. Cambridge University Press, 1999. ISBN 978-0-521-66350-2.

W. Swierstra. Heterogeneous binary random-access lists. *J. Funct. Program.*, 30:e10, 2020. `https://doi.org/10.1017/S0956796820000064`.

CMOS 4-Bit Magnitude Comparator. Texas Instruments, 2003.

T. Williams and D. Rémy. A principled approach to ornamentation in ML. *Proc. ACM Program. Lang.*, 2(POPL):21:1–21:30, 2018. `https://doi.org/10.1145/3158109`.