# Rough Pearl: Manufacturing Cons-Cells

Pierre-Evariste Dagand, Pierre Letouzey, Ellenor Fatemeh Taghayor

# Rough Pearl: Manufacturing Cons-Cells

Pierre-Évariste Dagand, Pierre Letouzey, and Ellenor Fatemeh Taghayor

Université Paris Cité, CNRS, IRIF

What if we could compose some 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 *numerical representations*, which made it to the Functional Programming Pantheon through the seminal work of Okasaki and were recently celebrated at Collège de France by Leroy.

This work offers to develop a uniform presentation of these objects in dependent type theory. We abide by two overarching principles. First, we strive to provide a generic and uniform representation of this family of data-types. Second, we strive to exactly capture the data-logic of sizes and cardinality at play. It is in the act of balancing the two opposite forces of generality and fine-grained invariants that we hope to find a Pearl.

Year 3024. An extra-terrestrial vessel has been recovered from the confines of the Sagittarius Arm of the Milky Way. The vehicle is identified as “Uss Lvmh”, according to scriptures on the hull. Documents found on board suggest that this vessel was carrying away a wealthy clientele of happy-fews fleeing a dying planet called Earth. An in-depth study of the ship shows that this lost civilization had elaborate religious rituals (dubbed “Agile ceremony”, lead by “Scrum masters”) with a strong animist influence (a deity named “Coq” played a central role in their theology). They also mastered advanced programming concepts. Investigations reveal that the ship tragically veered off course due to a floating point error in a machine-checked procedure, exploiting a loss of subject reduction in their proof checker. An act of sabotage is not excluded at this point.

Like any sufficiently advanced civilization, their programming language is indistinguishable from a dependently-typed functional programming, as we use them here on Rama. This report aims at documenting some programming idioms found in their code base. We ask our readers to keep an open-mind, as notation and usage may seem alien. While the following results are part of the Ramaian programming folklore, we believe that the Earthlings cast them in a novel, interesting way.

This work is, in particular, concerned with numerical representations [Okasaki, 1999, Chapter 9]. A typical example of a numerical representation are the Peano natural numbers

$$
\begin{array}{l}
\mathsf{type}\ \mathsf{Nat} \triangleq \\
\quad \begin{array}{|rcl}
\bullet & : & \mathsf{Nat} \\
(n : \mathsf{Nat})\ \mathsf{1} & : & \mathsf{Nat}
\end{array}
\end{array}
$$

that correspond to unary numbers, that is numbers in base $1^k$ coefficiented by the digit $\mathsf{1}$. We remark that Earthlings wrote these numbers with the most significant bit first, *i.e.* we have

$$\begin{array}{llllll} \mathsf{Nat{\Rightarrow}}\mathbb{N} & ( \quad \bullet & 1 & 1 & 1 & ) \\ & = \quad 0 & + 1 \times 1^2 & + 1 \times 1^1 & + 1 \times 1^0 & \end{array}$$

However, let us not be fooled by the postfix notation: as for traditional lists, this definition gives constant-time access to the least significant bit (the rightmost digit), as expected.

From the structure of unary numbers, we derive a *data*-structure: lists, which is obtained by ornamenting [Mcbride, 2010, Dagand, 2017] each digit 1 at position $k$ with $1^k$ pieces of data

$$\begin{array}{l} \mathsf{type}\ \mathsf{List}\ (A : \star) \triangleq \\ \quad \begin{array}{|rcl} \bullet & : & \mathsf{List}\ A \\ (xs : \mathsf{List}\ A) \triangleright (a : A) & : & \mathsf{List}\ A \end{array} \end{array}$$

In particular, lists inherit the left-leaning orientation of unary numbers. We recover the traditional “cons” operator by flipping both arguments. The symbol “$\star$” was pronounced “Type” by some Earthlings while it was pronounced “Set” by some others. Either way, it represents the (properly stratified) collection of all types.

The relationship between lists and numbers is materialized through the obvious projection that forgets the extra data and only keeps the recursive structure

$$\begin{array}{lcl} \multicolumn{3}{l}{\mathsf{length}\ (xs : \mathsf{List}\ A) : \mathsf{Nat}} \\ \mathsf{length}\ \bullet & \triangleq & \bullet \\ \mathsf{length}\ (xs \triangleright a) & \triangleq & (\mathsf{length}\ xs)\ 1 \end{array}$$

A key insight of numerical representations is that a significant part of the implementation and complexity analysis of data-structure can be systematically lifted up from the underlying numerical representation. For instance, since unary numbers give constant time access to the least significant digit, we obtain constant-time “cons” (by the very definition of List). We can also check that we can implement a constant-time predecessor over unary numbers

$$\begin{array}{lcl} \multicolumn{3}{l}{\mathsf{pred}\ (n : \mathsf{Nat}) : \mathsf{Nat}} \\ \mathsf{pred}\ \bullet & \triangleq & \bullet \\ \mathsf{pred}\ (n\ 1) & \triangleq & n \end{array}$$

and blindly follow this guide to obtain the tail of a list, through copy-and-paste (most likely!) or more refined schemes [Dagand and McBride, 2014, Williams and Rémy, 2018]:

$$\begin{array}{lcl} \multicolumn{3}{l}{\mathsf{tail}\ (xs : \mathsf{List}\ A) : \mathsf{List}\ A} \\ \mathsf{tail}\ \bullet & \triangleq & \bullet \\ \mathsf{tail}\ (xs \triangleright \_) & \triangleq & xs \end{array}$$

Similarly, the recursive definition of addition of unary number

$$\begin{array}{lcl} \multicolumn{3}{l}{(m : \mathsf{Nat}) + (n : \mathsf{Nat}) : \mathsf{Nat}} \\ \bullet + n & \triangleq & n \\ (m\ 1) + n & \triangleq & m + (n\ 1) \end{array}$$

is in fact specifying the size of the data-structure obtained by the concatenation of two lists. The tail-recursive implementation of addition thus yields the tail-recursive rev_append function

$$\begin{array}{lcl} \multicolumn{3}{l}{\mathsf{rev\_append}\ (xs\ ys : \mathsf{List}\ A) : \mathsf{List}\ A} \\ \mathsf{rev\_append}\ \bullet\ ys & \triangleq & ys \\ \mathsf{rev\_append}\ (xs \triangleright a)\ ys & \triangleq & \mathsf{rev\_append}\ xs\ (ys \triangleright a) \end{array}$$

Note that, wearing our computer scientist hat, we chose a tail-recursive, accumulator-based implementation of Peano addition. Had we been wearing our mathematician hat, we would probably have defined a stack-happy variant of addition as

$$\begin{array}{lcl} (m : \mathsf{Nat}) + (n : \mathsf{Nat}) : \mathsf{Nat} & & \\ \bullet + n & \triangleq & n \\ (m\ 1) + n & \triangleq & (m + n)\ 1 \end{array}$$

which would directly yield a stack-happy, in-order append function over lists.

This approach allows decoupling the treatment of the data (here, singletons of $A$-elements) from the treatment of the structure (here, unary numbers). The latter is entirely handled at the numerical level while, on this example, the former is trivial.

Indeed, unary numbers enjoy a certain conceptual simplicity, offering reasoning principles that are familiar to all high-school students. However, this representation remains highly inefficient: they take up linear space rather than logarithmic space, as offered by traditional Arabic or binary numerals. It is somewhat straightforward to define a binary number representation:

$$\begin{array}{l} \mathsf{type}\ \mathsf{Bin} \triangleq \\ \quad \begin{array}{|rcl} \bullet & : & \mathsf{Bin} \\ (bs : \mathsf{Bin})\ 0 & : & \mathsf{Bin} \\ (bs : \mathsf{Bin})\ 1 & : & \mathsf{Bin} \end{array} \end{array}$$

Once again, binary numbers are *written* with the most significant digit first

$$\begin{array}{lcccccc} \mathsf{Bin}{\Rightarrow}\mathbb{N} & ( & \bullet & 1 & 0 & 1 & 1 & ) \\ & = & 0 & +1 \times 2^3 & +0 \times 2^2 & +1 \times 2^1 & +1 \times 2^0 & \end{array}$$

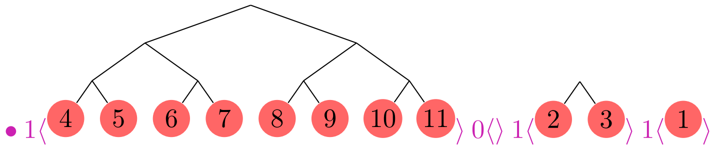
but we do get constant-time access to the least significant digit (rightmost constructor, following postfix notation).

One should however point out that this definition is not canonical: the number 0 can be coded as $\bullet$ but also $\bullet\ 0$ and in fact an infinite number of ways as $\bullet\ 0 \ldots 0$. Historical implementation of binary numbers strive to achieve canonicity [Agda standard library, Coq standard library], we shall see in a later section how this is handled in the context of numerical representations.

Random access-lists [Okasaki, 1999] provide logarithmic lookup and functional update operations, in addition to the usual interface of lists. They can be obtained by ornamenting each digit $1$ at position $k$ with $1 \times 2^k$ pieces of data. For random-access lists, a suitable container for storing powers-of-2 elements would, for example, be leaf binary tree (complete binary tree where data is stored in the leaves). Similarly, each digit $0$ at position $k$ is decorated with $0 \times 2^k = 0$ pieces of data. If we are willing to trust the programmer in respecting the size invariants, we could define the following data-structure:

$$\begin{array}{l} \mathsf{type}\ \mathsf{RAL01}(A : \star) \triangleq \\ \quad \begin{array}{|rcl} \bullet & : & \mathsf{RAL01}\ A \\ (bs : \mathsf{RAL01}\ A)\ 0\langle\rangle & : & \mathsf{RAL01}\ A \\ (bs : \mathsf{RAL01}\ A)\ 1\langle(t : \mathsf{LeafBinaryTree}\ A)\rangle & : & \mathsf{RAL01}\ A \end{array} \end{array}$$

An 11-elements inhabitant of this type would be:



The promise of numerical representations is that defining the successor function for binary numbers

$$\begin{array}{lcl}
\mathsf{incr}\ (bs : \mathsf{Bin}) : \mathsf{Bin} & & \\
\mathsf{incr}\ \bullet & \triangleq & \bullet\, 1 \\
\mathsf{incr}\ (bs\ 0) & \triangleq & bs\ 1 \\
\mathsf{incr}\ (bs\ 1) & \triangleq & (\mathsf{incr}\ bs)\ 0
\end{array}$$

translates into a blueprint for cons-ing an element to a random-access list

$$\begin{array}{lcl}
\mathsf{cons}\ (a : A)(bs : \mathsf{RAL01}\ A) : \mathsf{RAL01}\ A & & \\
\mathsf{cons}\ a\ \bullet & \triangleq & \bullet\, 1\langle a\rangle \\
\mathsf{cons}\ a\ (bs\ 0\langle\rangle) & \triangleq & bs\ 1\langle a\rangle \\
\mathsf{cons}\ a_1\ (bs\ 1\langle a_2\rangle) & \triangleq & (\mathsf{cons}\ (\mathsf{link}\ a_1\ a_2)\ bs)\ 0\langle\rangle
\end{array}$$

The only act of creativity left consists in appealing to link to join two leaf binary trees of $2^k$ so as to build a concatenated $2^{k+1}$ data container. This is an act of data management, all the structural work being handled by the definition of incr.

As witnessed by our discussion, the data container is subject to strict cardinality constraints, which are uniquely determined by the meaning of the numerical representation. For instance, we can belabor the "semantics" of the $k$-th digit of a binary number as follows

$$\begin{array}{rcl}
\mathsf{Bin{\Rightarrow}}\mathbb{N}_k\ \bullet & = & 0 \\
\mathsf{Bin{\Rightarrow}}\mathbb{N}_k\ (bs\ 0) & = & \mathsf{Bin{\Rightarrow}}\mathbb{N}_{k+1}\ bs + 0 \times 2^k \\
\mathsf{Bin{\Rightarrow}}\mathbb{N}_k\ (bs\ 1) & = & \mathsf{Bin{\Rightarrow}}\mathbb{N}_{k+1}\ bs + 1 \times 2^k
\end{array}$$

Read as a specification, this tells us that data-structures built from binary numbers must not decorate the • constructor while ensuring that each non-zero digit at position $k$ is decorated with $2^k$ pieces of data. Note that the above definition of random-access list rely solely on the programmer's discipline to ensure this invariant.

The distrusting nature of Earthling one-percenters led them to find ways to enforce these invariants through the type system. Using nested types [Bird and Meertens, 1998], Hinze [2001] has shown how well-sized data containers could be encoded in languages of the ML family. Nested types rely on non-uniform parameters to carry their mission

$$\begin{array}{lrcl}
\mathsf{type}\ \mathsf{RAL01N}\ (A : \star) \triangleq & & & \\
\mid & \bullet & : & \mathsf{RAL01N}\ A \\
\mid & (bs : \mathsf{RAL01N}\ (A \times A))\ 0\langle\rangle & : & \mathsf{RAL01N}\ A \\
\mid & (bs : \mathsf{RAL01N}\ (A \times A))\ 1\langle(a : A)\rangle & : & \mathsf{RAL01N}\ A
\end{array}$$

For instance, a random-access list with 11 elements would be encoded as thus

$$\bullet\, 1\langle(((4,5),(6,7)),((8,9),(10,11)))\rangle\ 0\langle\rangle\ 1\langle(2,3)\rangle\ 1\langle 1\rangle$$

This approach fell into disuse as it forcefully smashes intricate data containers (leaf binary trees in this case) down to a hodgepodge of binary products, making proper algorithmic work on the underlying data container meaningless. It also suffers from poor support in a dependently-typed setting, despite heroic efforts [Montin et al., 2022] made in that direction.

Our explorations suggest that an idiomatic, type-theoretic treatment of correct-by-construction numerical representations may be within our grasps. The presentation given in this paper is inspired by various experiments in the Coq and Agda proof assistants, at various levels of maturity and completeness. We therefore do not have any pretense concerning any machine-checked result.

Our objective is to hint at a compositional approach to numerical data-structure design. It draws inspiration from the work of Hinze [2001], which used nested types to encode a variety of numerical representations. The present paper can be understood as an actualization of this research program in a dependently-typed setting. We believe that one can combine an off-the-shelf numerical representation, a suitable data-container and, with minimal and localized efforts, obtain a data-structure and its key operations. At any rate, our experiments so far have been encouraging.

# 1 A Menagerie of Numerical Representations

In the following, we shall cover a carefully chosen selection of numerical representations. Each illustrates a salient property of numerical representation, which we intend to capture in our generic representation given in Section 2.

## 1.1 Well-typed or well-formed binary random-access lists

Rather than resort to nested types, we can exploit inductive families, presented either as an inductive predicate $\mathsf{valid\text{-}RAL01} : (k : \mathbb{N})(bs : \mathsf{RAL01}\ A) \to \star$, namely

$$\mathsf{valid\text{-}RAL01}_k\ \bullet \qquad \frac{\mathsf{valid\text{-}RAL01}_{k+1}\ bs}{\mathsf{valid\text{-}RAL01}_k\ (bs\ 0\langle\rangle)} \qquad \frac{\mathsf{valid\text{-}RAL01}_{k+1}\ bs \qquad \mathsf{size}\ t = 2^k}{\mathsf{valid\text{-}RAL01}_k\ (bs\ 1\langle t\rangle)}$$

$$\boxed{\mathsf{valid\text{-}RAL01} \triangleq \mathsf{valid\text{-}RAL01}_0}$$

or as an inductive family $\mathsf{dRAL01} : (k : \mathbb{N})(A : \star) \to \star$, namely

$$\begin{array}{l}
\mathsf{type}\ \mathsf{dRAL01}_{(k:\mathbb{N})}(A : \star) \triangleq \\
\quad \begin{array}{|rcl}
\bullet & : & \mathsf{dRAL01}_k\ A \\
(bs : \mathsf{dRAL01}_{k+1}\ A)\ 0\langle\rangle & : & \mathsf{dRAL01}_k\ A \\
(bs : \mathsf{dRAL01}_{k+1}\ A)\ 1\langle(t : \mathsf{LeafBinaryTree}_{2^k}\ A)\rangle & : & \mathsf{dRAL01}_k\ A
\end{array}
\end{array}$$

$$\boxed{\mathsf{dRAL01} \triangleq \mathsf{dRAL01}_0}$$

Conceptually, these data-*logics* can be understood as the fusion of $\mathsf{RAL01{\Rightarrow}}\mathbb{N}$ onto the underlying numerical data-structure, in a way reminiscent from algebraic ornaments [Dagand, 2017]. Note that these indexed types are *not* defining size-indexed structures (unlike, say, the perennial vector, *i.e.* length-indexed lists). Indices here are used to count the number of digits starting from the least significant one.

**Take-away:** we ought to be able to capture size invariants, either through intrinsic encoding (inductive family) or extrinsic encoding (inductive predicates).

## 1.2 Zeroless binary random-access lists

As hinted at earlier, lacking canonicity makes for cumbersome definitions (*e.g.*, when defining equality). Interestingly, this problem shows up during algorithmic work as well, for different reasons. Sometimes, canonicity is frowned upon so as to avoid having increment and decrement enter a "resonant" mode, with carries rippling left and right. This is the motivation behind data-structures such as 2-3 trees [Hinze, 2018]. Other times, a strict, zero-less representation is adopted so as to improve the density of the data representation. For instance, 1-2 binary numbers [Hinze, 2001] is defined as

$$
\begin{array}{l}
\mathsf{type}\ \mathsf{Bin12} \triangleq \\
\quad \begin{array}{|rcl}
\bullet & : & \mathsf{Bin12} \\
(bs : \mathsf{Bin12})\ 1 & : & \mathsf{Bin12} \\
(bs : \mathsf{Bin12})\ 2 & : & \mathsf{Bin12}
\end{array}
\end{array}
$$

which, in turn, leads to 1-2 binomial random-access list

$$
\begin{array}{l}
\mathsf{type}\ \mathsf{RAL12}(A : \star) \triangleq \\
\quad \begin{array}{|rcl}
\bullet & : & \mathsf{RAL12}\ A \\
(bs : \mathsf{RAL12}\ A)\ 1\langle(t : \mathsf{LeafBinaryTree}\ A)\rangle & : & \mathsf{RAL12}\ A \\
(bs : \mathsf{RAL12}\ A)\ 2\langle(t_0 : \mathsf{LeafBinaryTree}\ A)\rangle\langle(t_1 : \mathsf{LeafBinaryTree}\ A)\rangle & : & \mathsf{RAL12}\ A
\end{array}
\end{array}
$$

where every constructor (aside from $\bullet$) play an informational role, each of them storing some A. This is unlike the $0\langle\rangle$ constructor of the $\mathsf{RAL01}$ type, which played no other role than encoding the size of the underlying structure (a role which is nonetheless crucial from an operational standpoint).

**Take-away:** we ought to be able to support arbitrary sets of digits, especially ones without a zero.

## 1.3 Skew binary numbers/binomial heaps

An alternative approach to dispose of informationally-trivial 0s consists in adopting a "run-length encoded" presentation. Rather than having a constructor for the digit 0, one can decorate each non-zero digit with the number of *preceding* zeros. Alternatively, one could adopt a "sparse" representation, maintaining a set of pair of indices and coefficients for non-zero digits. From a type-theoretic standpoint, a sparse representation is trickier to deal with, for example to implement equality (*e.g.*, two representations are equal up to the order of the set of indices, opening the Doors of Setoid Hell[1]).

Myers' [Myers, 1983] skew binomial heap provides an historical example of run-length encoded representation, using a skew binary $(2^{k+1} - 1)$ base:

$$
\begin{array}{l}
\mathsf{type}\ \mathsf{SBin} \triangleq \\
\quad \begin{array}{|rcl}
\bullet & : & \mathsf{SBin} \\
(bs : \mathsf{SBin})\ 1 \overset{(c:\mathbb{N})}{\ldots} & : & \mathsf{SBin} \\
(bs : \mathsf{SBin})\ 2 \overset{(c:\mathbb{N})}{\ldots} & : & \mathsf{SBin}
\end{array}
\end{array}
$$

Here, the digits $1 \overset{c}{\ldots}$ and $2 \overset{c}{\ldots}$ carry an offset $c$. For instance, the digit $1 \overset{c}{\ldots}$ at position $k$ accounts for $1 \times (2^{(c+k)+1} - 1)$ elements, *i.e.* offsetting the base interpretation by $c$.

Note that, once again, this representation is not canonical. For example, $\bullet\, 2 \overset{1}{\ldots} 1 \overset{0}{\ldots}$ and $\bullet\, 2 \overset{0}{\ldots} 2 \overset{0}{\ldots}$ both denote the natural number 15. However, there exists a strict subset of skew binary numbers that is closed under increment, decrement and any other numerical operation one needs to implement a binomial heap. Moreover and amazingly, increment and decrement happen in *constant-time* on these canonical skew binary numbers: it is enough to look at the first 2 least significant digits to propagate the carry.

The normal form can be expeditiously characterized as "maybe start with a $2 \overset{c_0}{\ldots}$ followed by any, possibly empty, number of $1 \overset{c_i}{\ldots}$". Piponi and Yorgey [2015] indicates how one could

[1] As soon as there are multiple *equivalent* representations of a number, we are condemned to show that all our operations over these representations produce equivalent results. In practice, this adds a significant burden to an implementation effort.

systematically turn such a regular expression into an indexing discipline for SBin and, by extension, its corresponding numerical representation. It is also straightforward to work it out by hand in this case.

**Take-away:** we ought to be able to enforce canonicity of the numerical representation, either through intrinsic encoding (inductive family) or extrinsic encoding (inductive predicates). Failing that, some operations could not be implemented with the proper complexity.

## 1.4 Magic skew binary, or "These Numbers All Go to Eleven"

Our investigation lead us to stumble upon a mesmerizing numbering system, magic skew binary [Elmasry et al., 2012]. Based on the skewed binary base $(2^{k+1} - 1)$, magic skew binary use digits between 0 and 4. As for Myers' skew binary, only a strict subset of these numbers can be reached through the successor function. This subset has been characterized by Elmasry et al. [2012] as the union of 16 abstract forms: the successor function incr takes magic numbers from one form to another according to the following transitions:

$$\begin{tikzcd}
\bullet \arrow[r, "\mathsf{incr}"] & \bullet 1 \arrow[r, "\mathsf{incr}"] & \bullet 2 \arrow[r, "\mathsf{incr}"] & \bullet 01 \arrow[r, "\mathsf{incr}"] & \bullet 11 \arrow[r, "\mathsf{incr}"] & \tau 12 \arrow[r, "\mathsf{incr}"] & \tau 20 \arrow[r, "\mathsf{incr}"] & \tau 21 \arrow[d, "\mathsf{incr}"] \\
 & & & & & & & \beta 22 \arrow[d, "\mathsf{incr}"] \\
 & & & & & \gamma 11 \arrow[uu, "\mathsf{incr}"] \arrow[d, "\mathsf{incr}"'] & \gamma 30 \arrow[l, "\mathsf{incr}"'] & \beta 30 \arrow[l, "\mathsf{incr}"'] \\
 & & & & & \psi 32 \arrow[d, "\mathsf{incr}"'] & & \\
 & & & & & \psi 40 \arrow[r, "\mathsf{incr}"'] & \alpha 13 \arrow[r, "\mathsf{incr}"'] & \alpha 21 \arrow[uuu, bend right, "\mathsf{incr}"']
\end{tikzcd}$$

The 5 symbolic states $\alpha$, $\beta$, $\gamma$, $\psi$ and $\tau$ can themselves be described as regular expressions (to be read from right-to-left, in keeping with our notations)

$$\begin{aligned}
\tau &= 2^* \mid 12^* \\
\alpha &= \gamma 12^* \\
\beta &= 2^* \mid \tau 12^* \mid \psi 32^* \\
\gamma &= 1 \mid \tau 2 \mid \beta 3 \mid \psi 4 \\
\psi &= \gamma 0 \mid \alpha 1
\end{aligned}$$

For instance, the canonical representation of the number 10 amounts to

$$\begin{array}{rcccccc}
\mathsf{Magic}{\Rightarrow}\mathbb{N} & ( & \bullet & 1 & 0 & 3 & ) \\
= & & 0 & +1 \times (2^{2+1} - 1) & +0 \times (2^{1+1} - 1) & +3 \times (2^{0+1} - 1) &
\end{array}$$

This numerical representation offers a proving ground to the various representational choices presented earlier. For instance, the 16 abstract forms could certainly (though painfully!) be turned into an inductive family of canonical-by-construction magical binary numbers. However, we are still left with implementing the successor function incr. In Elmasry et al. [2012], this is achieved through a two-step process "(1) increment and denormalize, then (2) fix and renormalize". We have formalized and proved the correctness of this process over an extrinsic presentation in Coq. A canonical-by-construction, intrinsic representation calls for a radically different approach. Too radical to be found before the Jfla deadline.

**Take-away:** in some circumstances, we ought to have access to both canonical and non-canonical representations, operations over the former being implemented in terms of the latter.

## 2 Unified Numerical Representation

We can summarize our findings by offering a single, unified numerical representation. In our journey, we have identified 4 ingredients:

$$\begin{array}{rl} \mathsf{base} : & \mathbb{N} \to \mathbb{N} \\ \mathsf{digits} : & \star \\ \mathsf{T} : & \star \to \star \\ \mathsf{size\text{-}T} : & \mathsf{T}\ A \to \mathbb{N} \end{array}$$

We must first choose a numerical base. We have seen unary, $1^k$, binary, $2^k$, as well as skew binary $2^{k+1} - 1$. We must then choose a set of digits, *i.e.* the coefficients by which the base is multiplied. We have seen the singleton $\{1\}$, the sets $\{0, 1\}$, $\{1, 2\}$ or even $\{0, 1, 2, 3, 4\}$. We must choose a data container T, for which we shall ensure – through programmer's discipline – or enforce – through proofs or type-checking – that the size-T of data container follow the specification set by base. We have witnessed the use of leaf binary trees and hinted at the use of binomial heaps.

Provided these ingredients, we can define a generic numerical representation:

$$\begin{array}{l} \mathsf{type}\ \mathsf{Num}\ (A : \star) \triangleq \\ \quad \begin{array}{|rcl} & \bullet & : & \mathsf{Num}\ A \\ & (bs : \mathsf{Num}\ A)\ [(d : \mathsf{digits})]\ \overset{(c:\mathbb{N})}{\ldots} & : & \mathsf{Num}\ A \end{array} \end{array}$$

If we further enforce that offsets c are equal to 0 everywhere, we obtain a dense representation. Otherwise, this representation offers a run-length encoding by default. Similarly, size-T invariants and, further, canonical forms have to be enforced through further indexing or predicates on top of this base definition. We conjecture that there exists a sufficiently big crank that, once turned, yields the desired data-structures.

Note that operations (such as increment/insertion, decrement/deletion, addition/concatenation, *etc.*) remain to be engineered on a case-by-case basis. The overall methodology consists in first implementing the numerical operation (*e.g.*, increment), which is solely concerned with the numerical data-structure, from which one builds up the corresponding operation over the numerical data-type (*e.g.*, insertion).

In this context, unary numbers/lists could be (over-)engineered as an instance of

$$\begin{array}{l} \mathsf{List\text{-}base}\ (k : \mathbb{N}) : \mathbb{N} \\ \mathsf{List\text{-}base}\ k \quad \triangleq \quad 1^k \\[1ex] \mathsf{type}\ \mathsf{List\text{-}digits} \triangleq \\ \quad \begin{array}{|lcl} 1 & : & \mathsf{List\text{-}digits} \end{array} \\[1ex] \mathsf{List\text{-}T}\ (A : \star) : \star \\ \mathsf{List\text{-}T}\ A \quad \triangleq \quad A \end{array}$$

Similarly, skew binary numbers/skew binomial heaps correspond to an instance of

$$\begin{array}{l} \mathsf{SkewBin\text{-}base}\ (k : \mathbb{N}) : \mathbb{N} \\ \mathsf{SkewBin\text{-}base}\ k \quad \triangleq \quad 2^{k+1} - 1 \\[1ex] \mathsf{type}\ \mathsf{SkewBin\text{-}digits} \triangleq \\ \quad \begin{array}{|lcl} 1 & : & \mathsf{SkewBin\text{-}digits} \\ 2 & : & \mathsf{SkewBin\text{-}digits} \end{array} \\[1ex] \mathsf{SkewBin\text{-}T}\ (A : \star) : \star \\ \mathsf{SkewBin\text{-}T}\ A \quad \triangleq \quad \mathsf{LeafBinaryTree}\ A \end{array}$$

## 3 Conclusion

The link between numerical representations and data-structures has been extensively explored by some of the giants of computer science [Knuth, 1998, Vuillemin, 1978]. Ever since Okasaki's seminal work [Okasaki, 1999], it has been an integral part of the functional programming folklore. Recently, Leroy gave a crisp exposition (in French) as part of his 2023 lecture series at College de France[2]. Following Hinze [2001], he showed how structural invariants and, in fact, the data containers could be built using nested types.

The present work was born out of the first author's inability to simply capture these invariants in type theory. Other type-theorists, such as Ko and Gibbons [2017] and Swierstra [2020], tackled specific instances of the pattern but a general treatment remained elusive. We believe that this research program can succeed if (1) we strictly decouple the various levels of indexing involved, (2) we define the semantics of the underlying numerical systems through a carefully crafted recursive definition. We hope to be able to provide further evidences at the meeting.

**Future work.** This document builds up on various piecemeal experiments in Agda and Coq: a first order of business will consist in aggregating these results in a single, coherent library in both languages, exploiting their respective strengths and weaknesses (*e.g.*, dependent types *vs.* inductive predicates). In the process, we will have to clarify whether these definitions are obtained by internally *instantiating* generic Agda or Coq definitions, or through external *meta-programming*. Early experiments suggest that, barring some reasonable amount of duplication, a purely internalized presentation could be within reach.

A key motivation for the study of numerical representation lies in the promise of a compositional treatment of algorithmic complexity of data-structure operations, which would be decomposed as the cost of operations on the underlying data-container combined with the cost of arithmetic over the numerical representation. It would be interesting to fulfill this claim through machine-checked complexity proofs [Guéneau et al., 2018].

Our presentation has threaded very lightly around the topic of data containers. Being subject to strict size constraints, it is extremely tempting to use inductive families to enforce these invariants. However, there is a significant impedance mismatch to be solved at this interface. On the one hand, a numerical representation will want to impose precise cardinality constraint. On the other hand, typical data containers are indexed by more structural information (such as the height of a tree), which is only sometimes and indirectly related to the size (*e.g.*, log of the size for a leaf binary tree). Vectors and Braun trees [Okasaki, 1997] (which were suggested to us by Jean-Christophe Filliâtre) are among the few data-structures to be indexed by their actual size (in unary for vectors, in binary for Braun trees).

Conversely, numerical representations could be indexed by their size, as this is a standard case of reornamentation [Dagand, 2017]. However, it is not clear how exacting it would be to program with these definitions. Ultimately, we would certainly want our numerical representations to also satisfy the requirement for being data containers.

Finally, a grand challenge, which reaches beyond the grasps of our functional programming toolbox, would be to provide techniques and tools to identify the canonical forms of number systems. Judging by the work of Knuth [1998], Myers [1983] and Elmasry et al. [2012], these canonical forms are traditionally found by manually iterating the successor function starting from 0 and proving, after the fact, that these abstract forms are stable. Magic binary skew numbers offer a humbling testimony to the intellectual power of some of our colleagues.

**Acknowledgments.** Our interest in these questions was rekindled thanks to inspiring discussions with Catherine Dubois, Amélie Ledein, and Mathieu Montin. We are grateful to them for their time and enthusiasm.

[2] `https://www.college-de-france.fr/fr/agenda/cours/structures-de-donnees-persistantes/systemes-de-numeration-et-types-non-reguliers`

# References

Agda standard library. Binary numbers. `https://github.com/agda/agda-stdlib/blob/master/src/Data/Nat/Binary/Base.agda#L29-L32`.

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. doi: 10.1007/BFb0054285. URL `https://doi.org/10.1007/BFb0054285`.

Coq standard library. Binary numbers. `https://github.com/coq/coq/blob/master/theories/Numbers/BinNums.v#L21-L24`.

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

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

A. Elmasry, C. Jensen, and J. Katajainen. Two skew-binary numeral systems and one application. *Theory Comput. Syst.*, 50(1):185–211, 2012. doi: 10.1007/s00224-011-9357-0. URL `https://doi.org/10.1007/s00224-011-9357-0`.

A. Guéneau, A. Charguéraud, and F. Pottier. A fistful of dollars: Formalizing asymptotic complexity claims via deductive program verification. In A. Ahmed, editor, *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, volume 10801 of *Lecture Notes in Computer Science*, pages 533–560. Springer, 2018. doi: 10.1007/978-3-319-89884-1\_19. URL `https://doi.org/10.1007/978-3-319-89884-1_19`.

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

R. Hinze. On constructing 2-3 trees. *J. Funct. Program.*, 28:e19, 2018. doi: 10.1017/S0956796818000187. URL `https://doi.org/10.1017/S0956796818000187`.

D. E. Knuth. *The art of computer programming, Volume II: Seminumerical Algorithms, 3rd Edition.* Addison-Wesley, 1998. ISBN 0201896842. URL `https://www.worldcat.org/oclc/312898417`.

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

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

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. doi: 10.4204/EPTCS.360.2. URL `https://doi.org/10.4204/EPTCS.360.2`.

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

C. Okasaki. Three algorithms on braun trees. *J. Funct. Program.*, 7(6):661–666, 1997. doi: 10.1017/s0956796897002876. URL `https://doi.org/10.1017/s0956796897002876`.

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

D. Piponi and B. A. Yorgey. Polynomial functors constrained by regular expressions. In R. Hinze and J. Voigtländer, editors, *Mathematics of Program Construction - 12th International Conference, MPC 2015, Königswinter, Germany, June 29 - July 1, 2015. Proceedings*, volume 9129 of *Lecture Notes in Computer Science*, pages 113–136. Springer, 2015. doi: 10.1007/978-3-319-19797-5_6. URL `https://doi.org/10.1007/978-3-319-19797-5_6`.

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

J. Vuillemin. A data structure for manipulating priority queues. *Commun. ACM*, 21(4): 309–315, 1978. doi: 10.1145/359460.359478. URL `https://doi.org/10.1145/359460.359478`.

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