VERIFIED PURELY FUNCTIONAL CATENABLE REAL-TIME DEQUES

JULES VIENNOT [a], ARTHUR WENDLING [b], ARMAËL GUÉNEAU [c], AND FRANÇOIS POTTIER [a]

[a] Inria Paris, France
e-mail address: jules.viennot@inria.fr, francois.pottier@inria.fr

[b] Tarides, France
e-mail address: art.wendling@gmail.com

[c] Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, LMF, France
e-mail address: armael.gueneau@inria.fr

Abstract. We present OCaml and Rocq implementations of Kaplan and Tarjan's purely functional, real-time catenable deques. The correctness of our Rocq implementation is machine-checked.

1. Introduction

Twenty-five years ago, Kaplan and Tarjan [KT99] established a striking result: there exist "purely functional, real-time deques with catenation". In other words, there exists a data structure that enjoys the following properties:

- This data structure represents a sequence of elements.
- It supports *push* and *pop*, which insert and extract one element at the front end, and *inject* and *eject*, which insert and extract one element at the rear end. In other words, it is a *deque*, a double-ended queue.
- It supports concatenation, *concat*.
- It is immutable, therefore persistent: none of the five operations modifies or destroys its argument.
- Each of the five operations has worst-case time complexity $O(1)$.

In this paper, we present *the first implementation* of Kaplan and Tarjan's catenable deques. This implementation is expressed in the purely functional subset of the OCaml programming language [OCaml]. Furthermore, we present *the first verified implementation* of Kaplan and Tarjan's catenable deques. This implementation is expressed in Gallina, the programming language of the Rocq proof assistant [Rocq]. Its correctness is stated and verified within Rocq; in other words, it is machine-checked.

LOGICAL METHODS IN COMPUTER SCIENCE
DOI:10.46298/LMCS-Volume/Paper Nr. missing

1.1. **Kaplan and Tarjan's paper.** Kaplan and Tarjan proceed in several steps. Section 4 of their paper presents a *deque*, which does not support *concat*; section 5 presents a *catenable steque*, which supports *concat* but not *eject*; section 6 presents the *catenable deque*, which supports all five operations. The catenable deque in section 6 uses the deque of section 4 as a building block. The catenable steque of section 5, on the other hand, is ultimately unused; its main purpose seems pedagogical.

At each step, Kaplan and Tarjan give a high-level English description of the data structure, and explain its design. They typically begin with a sketch of the general shape of the data structure, which forms a certain kind of tree. Then, they restrict this shape by imposing several invariants. One invariant imposes bounds on the sizes of certain "buffers". Another invariant, the *regularity* invariant, requires a certain *coloring* scheme to be respected, where colors are dictated by buffer sizes. We refer to this high-level description as the *tree representation*. (Kaplan and Tarjan do not introduce a specific name for it.)

Although the tree representation enables a broad understanding of the data structure and is pedagogically helpful, it does not allow achieving worst-case time complexity $O(1)$. To achieve this complexity, direct pointers to subtrees with certain color characteristics are needed. Thus, a lower-level description, which reflects how the data structure must actually be laid out in memory, is needed. Kaplan and Tarjan refer to it as the *pointer representation*.[1]

The pointer representation can be significantly more complex than the tree representation. Indeed, an informal explanation or a formal definition of the tree representation can be broken up in several stages: first, the shape of the data structure is defined; then an assignment of colors to nodes is introduced; finally, a regularity invariant, which imposes certain requirements on the manner in which nodes are colored, restricts the shape of the data structure. In the pointer representation, on the other hand, the shape of the data structure, the color assignment, and the regularity invariant influence each other: therefore, they must be defined simultaneously. This difficulty is not apparent in Kaplan and Tarjan's paper because they use informal language. However, in a typed programming language, such as OCaml or Rocq, this difficulty is significant. A large fraction of our paper is in fact devoted to presenting and explaining our type definitions.

After describing each data structure and its two representations, Kaplan and Tarjan devote their attention to proving that, in the pointer representation, each operation *can* be implemented. That is, they prove that each operation preserves the regularity invariant, which means that the result of this operation can be expressed in the pointer representation. This proof is expressed as English prose. The fact that each operation is correct (that is, produces a data structure that represents the desired sequence of elements) and has worst-case time complexity $O(1)$ is not explicitly proved: these checks are considered "straightforward".

In summary, although Kaplan and Tarjan's result represents a remarkable achievement, their paper does not provide any type definitions or any code. It is reasonable to assert that a type definition is implicit in their description of each data structure and that an algorithm is implicit in their proof that the invariant *can* be preserved by each operation. However, spelling out these type definitions and this code is extremely challenging—so much so that, in the twenty-five years that have elapsed since their paper appeared, no implementation of Kaplan and Tarjan's catenable deques has been published. We are aware of several attempts, which we discuss at the end of this paper (Section 10).

[1] In a purely functional programming language, every data structure is "pointer-based", that is, represented in memory using memory blocks and pointers. In our OCaml and Rocq implementations, Kaplan and Tarjan's "pointer representation" is just an inductive data structure.

1.2. **Applications.** The reader may wonder about the practical applications of persistent catenable deques. It seems difficult to provide compelling examples. Persistent sequences of characters, or *strings*, are obviously very useful, but a full-featured "string" data structure must offer many operations that catenable deques do not support, such as random access (that is, reading or updating the i^{th} element of a string) or extracting substrings. In the algorithms literature, to the best of our knowledge, only a few papers need a persistent catenable deque data structure. One example is Demaine, Langerman, and Price's confluently persistent tries [DLP10]. Even in situations where such a data structure is needed, instead of Kaplan and Tarjan's purely functional catenable deques, one might prefer to use Kaplan, Okasaki, and Tarjan's simple persistent catenable deques [KOT00], which have internal mutable state, and whose time complexity guarantee is weaker, as it is amortized over a sequence of operations. These simpler deques have been implemented in OCaml and verified using Rocq and Iris by Ponsonnet and Pottier [PP26].

1.3. **This paper.**

1.3.1. *OCaml implementations.* Our OCaml implementations take advantage of OCaml's strong type discipline in several ways. Where possible, we exploit generalized algebraic data types (GADTs) to express data structure invariants: for example, we index several types with colors and encode constraints on colors within our type definitions. The OCaml type-checker is then able to verify, at compile time, that the code respects these invariants. Furthermore, also at compile time, it identifies provably dead branches in case analyses, and lets us omit these branches.

However, the expressive power of GADTs is limited: for example, properties that involve size arithmetic cannot easily be expressed. In one such case (Section 5.1), we exploit an abstract type with a phantom type parameter to enforce a form of one-sided security: inside the abstraction barrier, runtime assertions of the form "this cannot happen" ensure that the invariant is established; outside of the abstraction barrier, the OCaml type-checker statically verifies that the invariant is preserved.

Quite remarkably, our OCaml implementations of deques and catenable deques do not involve any recursive function definitions: there is no occurrence of **`let rec`** in the code. Thus, the fact that every operation has worst-case time complexity $O(1)$ is literally obvious. This is rather mind-boggling: although these data structures are described by complex recursive types, they are updated by non-recursive operations. An intuitive explanation for this phenomenon is that, by design, these data structures contain direct pointers to the places where work will be needed next. (An analogous and much simpler example is an implementation of a stack as a linked list. It is a recursive data structure. Yet, its "push" and "pop" operations are not recursive, because they access only the top of the stack.) This is not our contribution: it is Kaplan and Tarjan's magic. It is perhaps more clearly visible in our paper than in theirs because we offer executable code whereas they describe algorithms in prose.

1.3.2. *Rocq implementations.* Our Rocq implementations are inspired by our OCaml implementations, but differ in substantial ways. On the one hand, Rocq's type discipline offers much greater expressive power. This lets us statically enforce invariants and omit dead branches; thus there is no need for runtime assertions. On the other hand, to guarantee strong normalization, Rocq imposes restrictions that do not exist in OCaml. In particular, some of the algebraic data types that we define in OCaml cannot be literally ported to Rocq, because they are "truly nested" data types (Section 10.4), which, to the best of our knowledge, no current proof assistant accepts. In particular, we believe that they are not supported by Rocq, Agda, Lean, or Isabelle/HOL. We work around this problem by giving alternative type definitions, which Rocq accepts. These definitions carry extra "level" and "size" indices, but have simpler recursive structure.

To state the correctness of our Rocq implementations, a family of "model" functions is needed. A model function maps a data structure to the mathematical sequence of elements that this data structure represents. We represent a mathematical sequence as a Rocq list. We write $[]$ for the empty list, $[x]$ for a singleton list, and $xs +\!+\, ys$ for the concatenation of the lists xs and ys. A model function computes the *fringe* of a data structure. Model functions appear in statements of correctness. The correctness of a *push* operation, which inserts an element x into a data structure s, and which is expected to insert x "in front of" the elements of s, is expressed by the equation $\mathit{fringe}(\mathit{push}\ x\ s) = [x] +\!+\, \mathit{fringe}(s)$.

Our Rocq implementation includes the definition of a model function for each data structure and a statement of correctness for each operation. As claimed by Kaplan and Tarjan, the proofs of these correctness statements are "straightforward"—a fairly simple matter of proving equations that involve fringes, empty lists, singleton lists, and list concatenations. Somewhat unexpectedly, the main technical challenge that we encounter lies not in these correctness proofs but in the very definition of the model functions. Indeed, Rocq requires us to prove that these definitions, which are recursive, are in some sense well-founded. Our first attempt is rejected; fortunately, we are able to find a reformulation that Rocq accepts. Our work not only tests the limits of Rocq's expressiveness, but also stresses the quality and robustness of its implementation. It takes a very long time (between minutes and tens of minutes) for the Rocq type-checker to accept some of our definitions.

In summary, somewhat unexpectedly, the main challenges that we encounter and overcome are formulating our inductive type definitions and the inductive definitions of our model functions so that they are accepted by Rocq. In other words, we find that describing a data structure can be more difficult than implementing operations on this data structure, and that stating a correctness property can be more difficult than proving this property!

We do not prove that our Rocq implementation has worst-case time complexity $O(1)$. There are two main reasons why we do not do so. First, such a claim cannot easily be stated, because Rocq does not have a notion of computational cost. Indeed, as two computations that have the same result are deemed equal, there is no way of distinguishing their costs. Second, our Rocq implementation involves "size" indices, which are natural numbers. However, to the best of our knowledge, Rocq does not allow the programmer to indicate that indices are computationally irrelevant, by which we mean that indices can be erased at runtime and that index computations have no cost. This issue is discussed in Section 9.2.

1.3.3. *Alternative approaches.* A reader may wonder whether, by using rich type definitions, involving nested algebraic data types as well as color, size, and level indices, we are making life needlessly difficult for ourselves. An alternative approach, both in OCaml and in Rocq, would be to define simpler, coarser types, where fewer invariants are expressed a priori. Instead, the desired invariants would be expressed and verified a posteriori.

Out of necessity, such coarser types would contain junk: that is, they would have inhabitants that are not well-formed data structures. Therefore our code would have to contain more dynamic tests and more dead branches where (in OCaml) one places a runtime assertion of the form "this cannot happen" or (in Rocq) one returns an arbitrary inhabitant of the function's result type. In case no such inhabitant can be manufactured (because the required type is abstract), the function's result type would have to be changed to an option type. These changes in the code seem aesthetically unpleasant. Furthermore, in this approach, it is up to the programmer to recognize which branches are reachable (so a valid result must be returned) and which branches are dead (so an error or a dummy result can be returned). This can be quite difficult.

To some extent, this approach can be compared with working in an untyped language. The programmer cannot rely on static type-checking to eliminate a class of programming mistakes and to automatically recognize a class of dead branches; she must instead rely on her own discipline and on runtime checks. Thus, the question of which approach to prefer is reminiscent of the "long and rather acrimonious debate" between proponents of typed languages and advocates of untyped languages [Rey85]. In this case, the debate is between simpler, weaker types and richer, stronger types.

Okasaki [Oka99, §10.1.1] discusses this question. He offers the example of the nested data type `type 'a seq = Nil | Cons of 'a * ('a * 'a) seq`. (We use OCaml syntax.) Okasaki uses Standard ML, where this data type can be defined, but where many operations on this type cannot be defined, because Standard ML does not have polymorphic recursion. A workaround is to collapse elements, pairs of elements, pairs of pairs of elements, and so on, into a data type: `type 'a ep = Elem of 'a | Pair of 'a ep * 'a ep`. This data type contains junk: it contains not only perfectly balanced pairs of pairs of ... of elements, but also arbitrary unbalanced binary trees. Then, a coarse definition of `'a seq` as a non-nested data type can be given: `type 'a seq = Nil | Cons of 'a ep * 'a seq`. Although this approach is workable, Okasaki gives three reasons for preferring the original nested data type: elegance, efficiency, and static detection of programmer errors. Therefore he pretends that Standard ML supports polymorphic recursion, and lets the reader translate to Standard ML without polymorphic recursion, if necessary. A modern reader is likely to prefer a more expressive language, such as Haskell, OCaml, or Scala.

In this paper, we follow Okasaki's lead and prefer to use richer, more accurate types. During our first attempt at implementing Kaplan and Tarjan's data structures, we found the type-checker extremely helpful in detecting mistakes and dead branches. We do not claim that an alternative approach based on coarser types is unworkable. We believe that it can work. Certainly, now that we have been able to implement Kaplan and Tarjan's data structures, one could port our code to a simply-typed style. However, we would be surprised if this change made verification significantly easier. In Section 2, we briefly discuss the simply-typed approach and provide links to our repository, where, to a limited extent, we have experimented with this approach.

1.3.4. *Structure of this paper.* Following Kaplan and Tarjan, we proceed in several stages. First, as a warm-up, we present a data structure that represents a natural number in a redundant binary numbering system, and we implement this data structure in OCaml and in Rocq (Section 2). Then, we present Kaplan and Tarjan's deques, and implement them in OCaml and in Rocq (Section 3). Next, we move to Kaplan and Tarjan's catenable deques. We describe the tree representation and the pointer representation of this data structure (Section 4). We present our OCaml implementation of it (Section 5). We explain why Rocq's restrictions on inductive type definitions prevent a direct port of our OCaml code to

Rocq (Section 6). Fortunately, we are able to work around these restrictions by introducing "level" and/or "size" indices and by reformulating the definitions of certain "model" functions. Deques, which serve as a building block in the construction of catenable deques, must be indexed with levels and sizes. Therefore, we revisit our Rocq implementation of deques (Section 7). Then, we formulate inductive type definitions for catenable deques, indexed with levels, which Rocq accepts (Section 8). The paper ends with an experimental evaluation of the performance of our OCaml code, a discussion of several ways of executing our Rocq code (Section 9), and a review of the related work (Section 10).

1.3.5. *Structure of the code.* In this paper, we show very little code. Instead, we focus on the type definitions that describe the structure of deques and catenable deques. These type definitions, alone, are quite long and complex. Once these type definitions are given, the code is relatively straightforward. Our complete code is available online. A globe icon represents a link to our repository or to a specific file in this repository. In particular, the file `Signatures.v` sums up the types and operations that must be offered by an implementation of catenable deques, as well as the correctness properties that these operations must satisfy; and the file `Cadeque/Summary.v` proves that we have implemented these operations and established their correctness properties.

Although the code in our repository is split over several files, the paper is meant to be understood without worrying about file names. As a default rule, the code that is shown in each section of the paper is self-contained and does not refer to code that is shown in an earlier section.[2] As an exception to this rule, because our implementations of catenable deques depend on our implementations of non-catenable deques, we allow references from the former to the latter. These references are identified by the prefix `Deque.` For example, our OCaml implementation of cadeques (Section 5) contains several references to the module `Deque`, which represents our OCaml implementation of non-catenable deques (Section 3.2). Similarly, our Rocq implementation of catenable deques (Section 8) contains several references to the module `Deque`, which represents our Rocq implementation of non-catenable deques, indexed with levels and sizes (Section 7).

At the time of writing, our code is accepted by Rocq 8.19 but is rejected by Rocq 8.20 and by Rocq 9 with a type error that we do not understand. The error has to do with universe levels. We have reported this issue.[3]

2. Natural numbers: a redundant binary representation

2.1. **Concept.** To explain the fundamental mechanisms at play in their representations of *sequences*, Kaplan and Tarjan [KT99] point out a connection with representations of natural *numbers*. This connection is made clear in Section 3, where non-catenable deques are introduced. Through this lens, there is a loose analogy between the operation of inserting an element into a sequence and the operation of incrementing a number. Therefore, Kaplan and Tarjan ask: which representations of natural numbers support constant-time incrementation?

[2] Taking advantage of this convention, we allow distinct sections to define distinct objects by the same name. For example, the OCaml type named `chain` in Section 2, defined in Fig. 3, and the OCaml type named `chain` in Section 3.2, defined in Fig. 11, have nothing to do with one another.

[3] https://github.com/mattam82/Coq-Equations/issues/635

In the following, let us decorate the binary representation of a number with an overline, and let us write the least significant digit on the left. Thus, for example, the decimal number 8 equals the binary number $\overline{0001}$, and the decimal number 94 equals the binary number $\overline{0111101}$. The incrementation operation, or *successor*, is written S.

In the well-known *binary* representation, where a digit is a bit (that is, either 0 or 1), incrementing a number can require as little as a single bit flip, in the best case; yet, due to carry propagation, it can also require several bit flips. For example, incrementing 94 requires just one bit flip: $S\ (\overline{0111101})$ is $\overline{1111101}$; whereas incrementing 95 requires six bit flips: $S\ (\overline{1111101})$ is $\overline{0000011}$. In general, incrementing the binary representation of the number n has worst-case time complexity $O(\log n)$.

To guarantee constant-time incrementation, it is necessary to ensure that only a constant number of digits are affected by incrementation. Kaplan and Tarjan note that this is achieved by Clancy and Knuth's *redundant binary representation* (RBR) [CK77], together with a *regularity* invariant.

In this representation, a natural number is still represented in base 2, but there are *three* possible digits, namely 0, 1, and 2. For example, $\overline{021}$ is a representation of the number $0 \times 2^0 + 2 \times 2^1 + 1 \times 2^2$, which is 8. A number can have multiple representations: for instance, 94 can be written under the form $\overline{0111101}$, $\overline{011112}$, or $\overline{222221}$. Clearly, some representations are better than others. A representation that begins with the digit 0 or 1 can be incremented in constant time, by changing the first digit: for example, $S\ (\overline{1111101})$ is $\overline{2111101}$. A representation that begins with the digit 2 requires more work: for example, $S\ (\overline{222221})$ is $\overline{111112}$. Kaplan and Tarjan identify *regularity*, a property of RBRs that enables constant-time incrementation. Incrementing a regular representation is easy, and can be done in constant time. However, this operation can produce an irregular result. Thus, a way of transforming such a result into a regular representation, in constant time, is also needed.

Kaplan and Tarjan define regularity as follows.[4] Let us write (a representation of) a number as a list of digits $d_0 d_1 \ldots d_n$ where $d_i \in \{0, 1, 2\}$. Such a representation is *regular* if for every j such that $d_j = 2$, there exists an $i < j$ such that $d_i = 0$ and, for every $k \in (i, j)$, $d_k = 1$. This means that while scanning the list of digits from right to left (that is, from most significant to least significant digit), after one has encountered the digit 2, one must encounter the digit 0 before one encounters another 2 or runs out of digits. For example, the representations $\overline{0111101}$ and $\overline{011112}$, both of which denote the number 94, are regular; whereas $\overline{222221}$ is not.

Clearly, if $d_0 d_1 \ldots d_n$ is regular, then $d_0 \neq 2$. That is, the leftmost digit of a regular representation cannot be 2. Therefore, a regular representation can be incremented in constant time just by incrementing its leftmost digit. Naturally, this can break regularity. For example, incrementing $\overline{011112}$ yields $\overline{111112}$, which is not regular, as a right-to-left scan encounters the digit 2 but never encounters the digit 0.

Kaplan and Tarjan propose a simple way of transforming the result of an incrementation, in constant time, so as to guarantee that it is regular. They scan the digits from left to right, looking for the least significant digit d_i that is not 1. If d_i is 0, then there is nothing to do; this representation is already regular. If d_i is 2, then they set d_i to 0 and increment the next digit, d_{i+1}. We refer to this transformation as *regularization*.

4 Kaplan and Tarjan's redundant binary representation, which involves the digits 0, 1, and 2, is reminiscent of skew binary numbers [Mye83] [Oka99, §9.3]. However, Kaplan and Tarjan's notion of regularity differs from the canonical form that is imposed on skew binary numbers, where "only the lowest non-zero digit may be two".

For example, applying regularization to $\overline{111112}$ yields $\overline{1111101}$, which is regular, as desired. Incrementing $\overline{1111101}$ yields $\overline{2111101}$, which is irregular: regularizing it yields $\overline{0211101}$, which is regular. In summary, writing S for the combination of incrementation and regularization, we have the following sequence of operations:

$$\overline{011112} \xrightarrow{S} \overline{1111101} \xrightarrow{S} \overline{0211101}$$

It is not entirely obvious why regularization always produces a regular result! We verify this fact by implementing regularization in Rocq in Section 2.3.

Kaplan and Tarjan note that regularization "changes only a constant number of digits, thus avoiding explicit carry propagation". This suggests that regularization can be performed in constant time. A difficulty, however, is that regularization requires scanning the digits from left to right, looking for the least significant digit d_i that is not 1. Thus, a way of skipping a group of "1" digits, in constant time, is required. The simple-minded representation of a number as a linked list of digits, which one may call the *tree representation*, is not suitable. Instead, a more elaborate representation, the *pointer representation*, must be introduced.

In the pointer representation, consecutive "1" digits must be arranged into groups, allowing a group to be skipped in constant time. To describe this data structure, let us introduce some terminology. Following Kaplan and Tarjan, numeric digits are replaced with colors: the digit 0 becomes the *green* digit; the digit 1 becomes the *yellow* digit; the digit 2 becomes the *red* digit. We refer to green and red digits, along with the very first digit d_0, as *heads*. Following a head, a series of consecutive yellow digits forms a *body*. A head and its body constitute a *packet*. A number, or *chain*, is a list of packets. In other words, a chain is inductively defined as either the empty chain or a packet followed by a chain. Searching for the next non-"1" digit amounts to moving to the next packet: this representation allows this operation to be performed in constant time.

Let us depict a digit as a colored dot, whose color is green, yellow, or red. We use a large dot if the digit is a head, and a small dot otherwise, so every packet begins with a large dot. We separate two consecutive packets with some space. Thus, the numbers that we have used in the previous examples are depicted as follows:

$$\overline{011112} \xrightarrow{S} \overline{1111101} \xrightarrow{S} \overline{0211101}$$

We assign colors not only to digits, but also to packets and chains. A packet takes on the color of its head. A chain takes on the color of its first packet. (The empty chain is green.) We remark that a head can be yellow only if it is the first digit, d_0. Thus, if a chain contains a yellow packet, then this must be the first packet. Using these conventions, regularity can be defined as a set of simple constraints on colors, shown in Fig. 1.

Because the color of a chain is the color of its first packet, the constraint "a green packet must be followed with a green or red chain" (R1) could also be written "a green packet must be followed with a green or red packet". A similar remark can be made about constraints (R2) and (R3). Similarly, (R4) could be written under the form "the top packet must be green or yellow". By the "top packet", we mean the root packet, or, in the above picture, the leftmost packet.

(R1) A green packet must be followed with a green or red chain.
(R2) A yellow packet must be followed with a green chain.
(R3) A red packet must be followed with a green chain.
(R4) The top chain must be green or yellow.

FIGURE 1. Natural numbers: regularity constraints

2.2. **OCaml implementation.** We now describe the redundant binary representation of natural numbers, together with the regularity invariant, as OCaml type definitions. Then we implement incrementation and regularization as OCaml functions.

So far, we have described a packet as a pair of a head and a body, where a body is a possibly empty list of yellow digits. So, a packet was never empty, and its color was green, yellow, or red. Here, we prefer to merge "body" and "packet" into a single type. So, a packet is a possibly empty list of digits, and the empty packet must be viewed as uncolored.

As discussed in the introduction (Section 1.3.3), we wish to build the regularity constraints of Fig. 1 into our type definitions, so as to let the OCaml type-checker statically verify that the regularity invariant is maintained and that certain branches in the code are dead. An alternative approach, which is to use simpler and coarser types, is discussed in Section 2.3.

For this purpose, we must encode color information in our type definitions. The type of packets must be indexed with a color, so we can talk about packets of type `green packet`, `yellow packet`, and `red packet`. Furthermore, we must express our regularity constraints, some of which involve disjunctions: for example, a green packet must be followed with "a green or red chain" (R1). This is somewhat challenging, as the OCaml type system is based on the notion of type equality, and does not offer a natural way of expressing disjunction. To encode a disjunction of colors, or a set of colors, at the type level, we can imagine two approaches. One approach relies on OCaml's polymorphic variants,[5] whose static type discipline can express certain superset and subset constraints. However, polymorphic variants are a rather uncommon and advanced feature of OCaml, and do not have a counterpart in Rocq's type theory. The second approach, which we prefer, involves encoding colors using 3 bits: we use one bit per hue, where a hue is green, yellow, or red. Thus, the color "green" is encoded as 100, the color "yellow" as 010, and the color "red" as 001. The absence of a color can be encoded as 000. The disjunction "green or red or uncolored" can be understood as "not yellow" and can be encoded as ?0?, where "?" is a wildcard, that is, an anonymous type variable. These ideas result in the code of Fig. 2.

Building on this encoding of colors, we define a type of packets that is indexed with a color. In short, the type `'c packet` is the type of a packet whose color is `'c`. Thus, for example, a packet of type `red packet` must be a red packet, and a packet of type `uncolored packet` must be an uncolored packet. Furthermore, by taking advantage of type variables, it is possible to describe disjunctions of colors. In particular, as suggested above, the constraint "green or red or uncolored", which is synonymous with "not yellow", can be encoded as `_ * noyellow * _`. Thus, a packet of type `(_ * noyellow * _) packet` must be green or red or uncolored. Similarly, the constraint "yellow or uncolored", which is synonymous with "not green and not red", can be encoded by `nogreen * _ * nored`. Thus, a packet of type

[5] For documentation on polymorphic variants, please see the OCaml manual [OCaml] or the book Real World OCaml [MMH22].

```
(* A group of distinct types encode the presence or absence of a hue. *)
type somegreen  = SOME_GREEN
type nogreen    = NO_GREEN
type someyellow = SOME_YELLOW
type noyellow   = NO_YELLOW
type somered    = SOME_RED
type nored      = NO_RED
(* A color is a triple of three hue bits. *)
type green      =  somegreen *   noyellow *   nored
type yellow     =    nogreen * someyellow *   nored
type red        =    nogreen *   noyellow * somered
type uncolored  =    nogreen *   noyellow *   nored
```

FIGURE 2. Natural numbers (OCaml): type-level colors

```
type 'c packet =
  | Hole   :                              uncolored packet
  | GDigit : (nogreen * _ * nored) packet -> green  packet
  | YDigit : (nogreen * _ * nored) packet -> yellow packet
  | RDigit : (nogreen * _ * nored) packet -> red    packet
                        (* in each case, next packet must be yellow or uncolored *)

type (_, _) regularity =             (* parameters: packet color and chain color *)
  | G : (green , _ * noyellow * _) regularity (* next chain must be green or red *)
  | Y : (yellow,           green) regularity
  | R : (red   ,           green) regularity

type 'c chain =
  | Empty :                                                 green chain
  | Chain : ('c1, 'c2) regularity * 'c1 packet * 'c2 chain -> 'c1 chain

type number =
  | T : (_ * _ * nored) chain -> number   (* a number is a chain that is not red *)
```

FIGURE 3. Natural numbers (OCaml): types for packets, chains, and numbers

`(nogreen * _ * nored) packet` must be yellow or uncolored. This constraint appears in our definition of the type `packet`.

The definitions of the types `packet`, `chain`, and `number` appear in Fig. 3. They are generalized algebraic data types (GADTs) [XCC03].

The type `packet` defines a packet as a list of colored digits. There is a constructor for each kind of digit (green, yellow, and red) and a constructor for the empty packet, also known as a *hole*. An empty packet is uncolored; a nonempty packet takes on the color of its first digit. Only the head of a packet can be a green digit or a red digit; every subsequent digit must be yellow. This is expressed by letting the constructors `GDigit`, `YDigit`, and `RDigit` require a yellow or uncolored packet.

```
1  (* Turn a red chain into a green chain. *)
2  let green_of_red : red chain -> green chain = function
3    | Chain (R, RDigit Hole, Empty) ->
4        Chain (G, GDigit (YDigit Hole), Empty)
5    | Chain (R, RDigit Hole, Chain (G, GDigit body, c)) ->
6        Chain (G, GDigit (YDigit body), c)
7    | Chain (R, RDigit (YDigit body), c) ->
8        Chain (G, GDigit Hole, Chain (R, RDigit body, c))
9
10 (* Turn a green or red chain into a green chain. *)
11 let ensure_green : type g r. (g * noyellow * r) chain -> green chain = fun c ->
12   match c with
13   | Empty           -> Empty
14   | Chain (G, _, _) -> c
15   | Chain (R, _, _) -> green_of_red c
16
17 (* Increment a number and perform regularization. *)
18 let succ : number -> number = function
19   | T Empty ->
20       T (Chain (Y, YDigit Hole, Empty))
21   | T (Chain (G, GDigit body, c)) ->
22       T (Chain (Y, YDigit body, ensure_green c))
23   | T (Chain (Y, YDigit body, c)) ->
24       T (green_of_red (Chain (R, RDigit body, c)))
```

FIGURE 4. Natural numbers (OCaml): regularization and incrementation

The type `regularity` defines the relationship between the color of a packet and the color of the subsequent chain. The three constructors that appear in its definition reflect the regularity constraints (R1)–(R3).

The type `chain` defines a chain as a list of packets. The empty chain is green. In a nonempty chain, the color `'c1` of the packet and the color `'c2` of the subsequent chain must obey a regularity constraint. A look at the definition of the type `regularity` shows that the color `'c1` must be be green, yellow, or red: it cannot be uncolored. The color of a nonempty chain is the color of its first packet. As a result, a chain is always green, yellow, or red; it cannot be uncolored. This explains our comment on line 9. Although the constraint `_ * noyellow * _` may seem to allow the next chain to be "green or red or uncolored", we know that a chain cannot be uncolored. Therefore we write that "the next chain must be green or red".

The type `number` defines a number as a chain that is not red. Therefore, a number is a green or yellow chain. This encodes the regularity constraint (R4).

Based on these type definitions, regularization and incrementation can be implemented as OCaml functions. These functions appear in Fig. 4.

The function `green_of_red` converts a red chain to a green one in the way described by Kaplan and Tarjan (Section 2.1), that is, by turning the first digit (which must be red) into a green one and by incrementing the following digit. Three cases arise, as the following digit might be absent, green, or yellow. The first case (lines 3–4) is straightforward. In the second case (lines 5–6), two packets are merged into one. The OCaml type-checker verifies that `GDigit (YDigit body)` is a well-formed green packet and that the chain `c` is allowed to follow a green packet. In the third case (lines 7–8), one packet is split into two. The OCaml type-checker verifies that `GDigit Hole` and `RDigit body` are well-formed packets and that the chain `c` is allowed to follow a red packet.

We are fortunate that the type-checker performs these checks (so we cannot inadvertently break regularity) and that it performs them silently for us (so the presence of these checks does not clutter the code). Furthermore, the OCaml type-checker verifies that all cases are covered: that is, the patterns on lines 3, 5, and 7 cover all possible forms of red chains. In other words, the type-checker verifies that all omitted cases are dead (contradictory). This is also an invaluable feature: here and especially in the forthcoming sections of this paper, it would be extremely difficult (in fact, we dare say, almost impossible) to ensure coverage without mechanical help.

The function `ensure_green` transforms a green or red chain into a green one.[6] The function `succ` increments a number and performs regularization, if necessary, to maintain regularity.

It is worth noting that none of the functions in Fig. 4 is recursive: no **`let rec`** keyword appears. Therefore, the worst-case time complexity of these functions is constant. The same striking remark holds about the more complex data structures presented in this paper: *no recursive functions* appear in the implementation of these data structures.

2.3. **Rocq implementation.** We now repeat the exercise of the previous section, this time inside Rocq. That is, we encode the redundant binary representation of natural numbers and the regularity invariant as Rocq type definitions, and we implement incrementation and regularization as Rocq functions.

We begin by encoding colors in Rocq. A direct transliteration of the encoding that was used in the previous section (Section 2.2) would lead us to define a hue bit ("some green", "no green", and so on) as an object of type `Type`. A color, which is a tuple of three hue bits, would then have type `Type * Type * Type`. However, that would be unnatural and needlessly complex. Because Rocq has true dependent types, there is no need for hue bits to be encoded as objects of type `Type`. Instead, we define each of the three hues, separately, as an inductive type with two constructors (Fig. 5). Then, we define a color as a combination of three hue bits, namely, a green bit, a yellow bit, and a red bit. Instead of a tuple, we use an inductive type with one constructor, `Mix`.

Then, we define packets, regularity, chains, and numbers, in Rocq, as shown in Fig. 6. These definitions are isomorphic to our earlier OCaml definitions (Fig. 3), but colors now have type `color` instead of being encoded as types. We let the type `regularity` inhabit the sort `Type`, as opposed to `Prop` (line 8). This means that (during the extraction of Rocq to OCaml) a regularity witness cannot be erased; it must exist at runtime. In return, this allows us to perform case analyses on regularity witnesses, not just in proofs, but also in our code. Although we currently exploit this ability, we believe that, with some extra effort, we could avoid depending on it, thereby allowing regularity witnesses to be erased.

As discussed in the introduction (Section 1.3.3), we choose to define packets, chains, and numbers as indexed types, and we build a regularity requirement into the definitions of

6 The type of `ensure_green` involves a universal quantification on the type variables `g` and `r`. Unfortunately, wildcards, which would be slightly more readable, cannot be used here.

```
(* Each of the three hues is an inductive type with two constructors. *)
Inductive green_hue  := SomeGreen  | NoGreen.
Inductive yellow_hue := SomeYellow | NoYellow.
Inductive red_hue    := SomeRed    | NoRed.

(* A color is a triple of three hue bits. *)
Inductive color      := Mix : green_hue -> yellow_hue -> red_hue -> color.
Notation green       := (Mix SomeGreen   NoYellow   NoRed).
Notation yellow      := (Mix   NoGreen SomeYellow   NoRed).
Notation red         := (Mix   NoGreen   NoYellow SomeRed).
Notation uncolored   := (Mix   NoGreen   NoYellow   NoRed).
```

FIGURE 5. Natural numbers (Rocq): types for colors

```
Inductive packet : color -> Type :=
  | Hole       : packet uncolored
  | GDigit {y} : packet (Mix NoGreen y NoRed) -> packet green
  | YDigit {y} : packet (Mix NoGreen y NoRed) -> packet yellow
  | RDigit {y} : packet (Mix NoGreen y NoRed) -> packet red.
         (* next packet must be yellow or uncolored *)

Inductive regularity : color -> color -> Type :=
  | G {g r} : regularity green  (Mix g NoYellow r)
                                (* next chain must be green or red *)
  | Y       : regularity yellow green
  | R       : regularity red    green.

Inductive chain : color -> Type :=
  | Empty                                                       : chain green
  | Chain {C1 C2 : color} : regularity C1 C2 -> packet C1 -> chain C2 -> chain C1.

Inductive number : Type :=
  | T {g y} : chain (Mix g y NoRed) -> number.
                    (* a number is a chain that is not red *)
```

FIGURE 6. Natural numbers (Rocq): types for packets, chains, and numbers

these types. In other words, we impose regularity *a priori*: a data structure that violates regularity cannot be constructed.

An alternative approach is to use simpler, coarser types, that is, to define packets, chains, and numbers as simple (non-indexed) inductive types, and to impose regularity *a posteriori*, as a property of packets, chains, and numbers. We have explored this approach, but do not describe it in the paper. (For the reader's benefit, a link is provided.) We find that neither approach seems significantly superior to the other.

In either approach, we find it extremely helpful to set things up so that (by exploiting regularity) Rocq is able to identify certain branches as dead and lets us omit dead branches. Furthermore, we find it essential to set things up so that Rocq does not require us to write a case analysis whose structure mimics the structure of the code. To achieve these two goals,

```
(* Turn a red chain into a green chain. *)
Equations green_of_red : chain red -> chain green :=
green_of_red (Chain R (RDigit Hole) Empty) :=
  Chain G (GDigit (YDigit Hole)) Empty;
green_of_red (Chain R (RDigit Hole) (Chain G (GDigit body) c)) :=
  Chain G (GDigit (YDigit body)) c;
green_of_red (Chain R (RDigit (YDigit body)) c) :=
  Chain G (GDigit Hole) (Chain R (RDigit body) c).
```

FIGURE 7. Natural numbers (Rocq): regularization, without a correctness claim

we use Rocq's *Equations* facility [SM19; Equations]. *Equations* is so named because it lets the user write a function definition in the form of a set of equations. However, this is not its most important aspect. More crucially, it allows pattern matching on indexed types, allows dead branches to be omitted,[7] and is able to extract "proof obligations" out of each equation separately, saving the user the trouble of writing a case analysis that mimics the structure of the equations.

The use of *Equations* is illustrated in Fig. 7, which presents a definition of the function `green_of_red`. We explicitly declare that this function expects a red chain as an argument. As a result, *Equations* recognizes that the three cases that are explicitly considered cover all possible situations; there is no need to explicitly consider and eliminate the remaining cases. Furthermore, we explicitly declare that `green_of_red` returns a green chain. This claim, too, is automatically verified by Rocq.

A fact that we have not expressed yet is the *correctness* of `green_of_red`. We are not content to know that the chain `green_of_red c` is green: we want to know that it represents the same natural number as the chain `c`. To express this fact, we must define which natural number is represented by a packet, a chain, or a number. In other words, we must define *model functions* that map each data structure to its model, a natural number.

These definitions appear in Fig. 8. The function `packet_nat` takes the model of the hole as an extra parameter: that is, `packet_nat pkt n` computes the model of the packet `pkt`, a natural number, under the assumption that the model of the hole is the natural number `n`. The definition of `packet_nat pkt n` is formulated in such a way that the number `n` is multiplied by 2^k where k is the length of the packet `pkt`. In the definition of `chain_nat`, at line 9, the parameter `n` is instantiated with `(chain_nat c)`: that is, the model of a chain is the model of its first packet, under the assumption that the hole represents the remainder of the chain.

It is worth noting that the model functions `packet_nat`, `chain_nat`, and `number_nat` are inductively defined. Although recursion is not needed in the code that operates on packets, chains, and numbers, induction is needed in the specification and proof of this code.

[7] As far as we can tell, as of today, *Equations* can "see" that a branch is dead, and allows this branch to be omitted, only if an unsatisfiable equation between indices appears in this branch. We heavily rely on this feature. If one can prove that a branch is dead via some other argument then this branch must still be provided. Its body can be a wildcard _. This gives rise to a proof obligation, whose proof can begin with `exfalso`. The user is then expected to provide a proof that this branch is dead.

```
Equations packet_nat {C : color} : packet C -> nat -> nat :=
packet_nat Hole n := n;
packet_nat (GDigit body) n := 0 + 2 * packet_nat body n;
packet_nat (YDigit body) n := 1 + 2 * packet_nat body n;
packet_nat (RDigit body) n := 2 + 2 * packet_nat body n.

Equations chain_nat {C : color} : chain C -> nat :=
chain_nat Empty := 0;
chain_nat (Chain _ pkt c) := packet_nat pkt (chain_nat c).

Equations number_nat : number -> nat :=
number_nat (T c) := chain_nat c.
```

FIGURE 8. Natural numbers (Rocq): model functions

Once the model function `chain_nat` has been defined, the correctness of the function `green_of_red` can be stated. Two approaches come to mind: this correctness statement can be expressed either *a posteriori*, as a stand-alone lemma, or *a priori*, as part of the definition of `green_of_red`. In the first approach, the correctness statement would take the following form:

```
Lemma green_of_red_correct :
  forall (c : chain red),
  chain_nat (green_of_red c) = chain_nat c.
Proof. ... Qed.
```

The proof must mimic the structure of the code; fortunately, a custom case analysis principle is automatically generated by *Equations* and helps reduce the redundancy between the code and the proof. In the second approach, which we adopt, no lemma is required: instead, at definition time, the function `green_of_red` is ascribed the dependent function type `forall (c : chain red), { c' : chain green | chain_nat c' = chain_nat c }`. This type claims that `green_of_red` transforms a red chain `c` into a green chain `c'` such that `c` and `c'` represent the same number. We refer to the equation `chain_nat c' = chain_nat c` as the *postcondition* of the function `green_of_red`. This approach integrates well with *Equations*: in each case, *Equations* automatically generates a proof obligation (that is, a goal) of the form `chain_nat c' = chain_nat c`, where `c` and `c'` are suitably instantiated. The proofs of these goals are provided by the user below the function's definition and (in many cases) can be automated via tactics.[8]

Our Rocq code for regularization and incrementation, with correctness claims, is shown in Fig. 9. Up to minor differences in syntax, the code is essentially the same as the OCaml code of Fig. 4. Each function carries a postcondition, which expresses

[8] The command `Obligation Tactic := <tactic>.` lets the user provide a tactic that is applied by *Equations* to every proof obligation. If this user-provided tactic is able to solve a proof obligation then this obligation is never presented to the user. Otherwise this obligation is presented to the user, who is expected to write a proof, delimited by `Next Obligation ... Qed.` In Fig. 9, every proof obligation is automatically solved by *Equations*'s default obligation tactic, so neither `Obligation Tactic` nor `Next Obligation` need be used.

```
(* ? x denotes an application of the subset type constructor to the witness x. *)
Notation "? x" := (@exist _ _ x _) (at level 100).

(* Turn a red chain into a green chain that represents the same number. *)
Equations green_of_red (c : chain red) :
  { c' : chain green | chain_nat c' = chain_nat c } :=
green_of_red (Chain R (RDigit Hole) Empty) :=
  ? Chain G (GDigit (YDigit Hole)) Empty;
green_of_red (Chain R (RDigit Hole) (Chain G (GDigit body) c)) :=
  ? Chain G (GDigit (YDigit body)) c;
green_of_red (Chain R (RDigit (YDigit body)) c) :=
  ? Chain G (GDigit Hole) (Chain R (RDigit body) c).

(* Turn a green or red chain into a green chain that represents the same number. *)
Equations ensure_green {g r} (c : chain (Mix g NoYellow r)) :
  { c' : chain green | chain_nat c' = chain_nat c } :=
ensure_green Empty :=
  ? Empty;
ensure_green (Chain G pkt c) :=
  ? Chain G pkt c;
ensure_green (Chain R pkt c) :=
  green_of_red (Chain R pkt c).

(* Turn a number n into a number n' that represents the successor of n. *)
Equations succ (n : number) :
  { n' : number | number_nat n' = S (number_nat n) } :=
succ (T Empty) :=
  ? T (Chain Y (YDigit Hole) Empty);
succ (T (Chain G (GDigit body) c))
  with ensure_green c => { | ? c' :=
  ? T (Chain Y (YDigit body) c') };
succ (T (Chain Y (YDigit body) c))
  with ensure_green (Chain R (RDigit body) c) => { | ? c' :=
  ? T c' }.
```

FIGURE 9. Natural numbers (Rocq): regularization / incrementation, with correctness claims

its correctness property. The functions `green_of_red` and `ensure_green` have postconditions of the form `chain_nat c' = chain_nat c`. The function `succ` has the postcondition `number_nat n' = S (number_nat n)`, where `S : nat -> nat` is the successor function for natural numbers: this means that `succ` correctly computes the successor of its argument.

Because the codomain of each function is a subset type `{ x : T | P x }`, the code must contain construction and deconstruction operations for the subset type.[9] We write `? x` for an application of the constructor of the subset type, `exist`, to the witness `x` and to a wildcard `_`. When this notation is used for construction, the wildcard gives rise to a proof obligation:

9 *Program* [Soz07], a predecessor of *Equations*, allowed implicit construction and deconstruction of subset types. *Equations* does not seem to allow this.

a proof of `P x` must be provided. When it is used for deconstruction, the wildcard stands for an anonymous proof of `P x`.

The **`with`** notation allows calling an auxiliary function and deconstructing its result. For example, on line 33, the variable `c'` is bound to the chain produced by the function call `ensure_green (Chain R (RDigit body) c)`. This can be understood as an alternative notation for a local definition.

3. Non-catenable deques

3.1. **Concept.** Kaplan and Tarjan [KT99, §4] introduce *non-catenable deques*, which we refer to as just *deques*. They support the four operations *push*, *pop*, *inject*, and *eject*, with worst-case time complexity $O(1)$. Deques serve as a stepping stone between natural numbers (Section 2) and catenable deques (Section 4). They are more complex than natural numbers, yet simpler than catenable deques, so, from a pedagogical point of view, it is useful to understand them before attempting to understand catenable deques. Furthermore, deques actually serve as a building block in the construction of catenable deques.

3.1.1. *Tree representation, without buffers.* When a deque's elements and buffers are abstracted away, the structure that remains is just a natural number in redundant binary representation, that is, a list of digits (Section 2). This is the deque's *tree representation*, without buffers. We also refer to it as the deque's *skeleton*. To grow or shrink this skeleton, one uses incrementation, decrementation, and regularization of natural numbers in RBR.

The skeleton of a deque of n elements is a representation of *some* natural number n' which may differ (and usually differs) from n. Indeed, as explained shortly after, each digit is expected to carry one or two buffers, which together can contain between 0 and 10 elements.

3.1.2. *Tree representation, with buffers.* Onto the skeleton, a number of elements are attached. More precisely, elements are stored in *buffers*, which are attached to the skeleton. A buffer is a sequence of 0 to 5 elements. We refer to the number of elements of a buffer as the *size* of this buffer. Because a deque must support inserting or extracting elements at either end, *two* buffers, a *prefix buffer* and a *suffix buffer*, are attached onto each digit. The last digit is special: it carries only one buffer. (We deviate slightly from Kaplan and Tarjan's presentation, which does not have such a special treatment of the last digit. We find that this deviation simplifies our code.) This is the deque's *tree representation*, with buffers.

An illustration of this structure appears in Fig. 10. Following Kaplan and Tarjan, we organize this drawing in a vertical manner. Each level corresponds to a colored digit. (In the drawing, the colors are omitted.) The top level is the least significant digit; the bottom level is the most significant digit. In the drawing, a horizontal line segment denotes a buffer. Each level (except the last level) consists of two buffers and a pointer to the next level. The last level consists of just one buffer.

To find out which sequence of elements is represented by a deque, the prefix buffers, which appear on the left-hand side of the illustration, must be read first, from top to bottom; then, the suffix buffers, which appear on the right-hand side, must be read from bottom to top. This is suggested by the curved arrow in Fig. 10. Thus, one can think of the sequence of elements as "folded in half" over the skeleton. The front and rear ends of the sequence are easily accessible, while the middle part of the sequence is deeply buried.

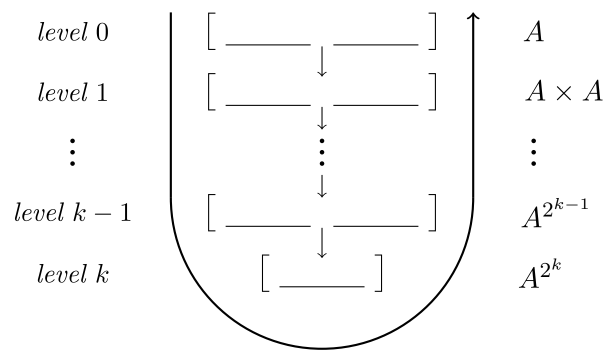


FIGURE 10. Tree representation of a deque, with buffers. The curved arrow indicates the logical order in which buffers are read. On the right are the types of the elements stored in the buffers.

3.1.3. *Recursive slowdown.* As one traverses the binary representation of a natural number, from least significant toward most significant digit, each digit carries a weight that is twice the weight of the previous digit. Kaplan and Tarjan apply a similar principle, known as *recursive slowdown*, to the elements that are stored in a deque's buffers. In a deque that holds elements of type A, the two buffers at the top level contain elements of type A; the two buffers at the next level contain elements of type $A \times A$; and so on. In general, the elements stored in the buffers at level i have type A^{2^i}. This is illustrated in the rightmost column of Fig. 10. To interpret a buffer as a sequence of elements of type A, one reads it from left to right and one expands each composite element of type A^{2^i} into a sequence of 2^i basic elements of type A.

3.1.4. *Color assignment.* In Section 2, a color (green, yellow, or red) was assigned to each digit in the representation of a natural number, based on the value of this digit. Here, each level in a deque is assigned a color, based on the sizes of its buffers, as follows. Kaplan and Tarjan define a mapping of sizes to colors:

- Sizes 2 and 3 are green; sizes 1 and 4 are yellow; sizes 0 and 5 are red.

This mapping allows assigning a color to a buffer, based on its size, that is, based on the number of elements that it contains.

An ordering on colors is defined: red < yellow < green. This means that red is worse than yellow, which itself is worse than green. Both extreme sizes, 0 and 5, are bad, because an element cannot be extracted out of an empty buffer and an element cannot be inserted into a buffer that is full.

Then, Kaplan and Tarjan assign colors to levels, based on the colors of their buffers. The color of a level is defined as the worst color of its two buffers, that is, the minimum color among the colors of its prefix buffer and suffix buffer. The last level, which contains just one buffer, is always green, regardless of the color of its buffer.

3.1.5. *Regularity.* With deques now rigorously defined, we return to the more general tree representation, corresponding to a natural number in RBR. By virtue of this representation, levels are grouped into packets and regularity constraints are imposed in the same way as in Section 2. The regularity constraints are unchanged; they can be found in Fig. 1.

3.1.6. *Pointer representation.* In the pointer representation, a deque is a *chain*, that is, a list of *packets*. This representation allows moving from one packet to its successor in constant time. Our OCaml and Rocq implementations (Sections 3.2 and 3.3) are based on it.

3.1.7. *Operations.* *push* and *inject* are analogous to incrementation in Section 2, while *pop* and *eject* are analogous to decrementation. Each of the four operations first naively inserts or removes one element, potentially making the chain irregular. Then, the chain is regularized. Both steps are carried out in constant time.

3.2. **OCaml implementation.** We first define the type of buffers, `('a, 'c) buffer`, as a generalized algebraic data type, with one constructor for each buffer size, from 0 to 5. This type carries two parameters: `'a` is the type of the elements of the buffer; `'c` is the color of this buffer. Its definition appears in Fig. 11.

We assign colors to buffers in a way that differs slightly from Kaplan and Tarjan's scheme, and is more flexible. Here is why. Thinking ahead about the way levels are colored, we wish to avoid the need to compute the minimum of two colors, because such a computation is not easily expressed, in OCaml, at the level of types. Instead, we make the type `buffer` covariant in the parameter `'c`. That is, we allow a buffer to be assigned a worse color than its true color. In other words, we allow a buffer that contains 1 or 4 elements to be viewed not only as a yellow buffer, but also, if desired, as a red buffer; and we allow a buffer that contains 2 or 3 elements to be viewed not only as a green buffer, but also, if desired, as a yellow buffer or a red buffer. This coloring scheme is more flexible than Kaplan and Tarjan's scheme, yet still lets us perform the four operations on deques in constant time in the worst case.

The type `('a, 'b, 'c) packet` (Fig. 11) is defined in a way that is reminiscent of the previous section: for comparison, see Fig. 3 in Section 2. There is still a constructor for the *hole*, that is, for the end of a packet. There is now a single constructor, `Packet`, for non-hole packets, irrespective of their color. This constructor carries a prefix buffer, a child packet, and a suffix buffer. As before, the child packet must be yellow or uncolored. As explained earlier, instead of computing the minimum of the colors of the two buffers, we require the two buffers to have the same color, namely `'c` (Fig. 11, lines 13 and 15). This color becomes the color of this packet (Fig. 11, line 16).

In `('a, 'b, 'c) packet`, the type parameter `'a` is the type of the elements of this packet. Thus, in a packet whose elements have type `'a`, the child packet has elements of type `'a * 'a` (Fig. 11, line 14). The type parameter `'b` is the type of the elements of the next packet in the chain. This is why the constructor `Hole` imposes an equality between `'a` and `'b` (Fig. 11, line 11): once the end of the current packet is reached, the type `'a` of the elements of the current packet must coincide with the type `'b` of the elements of the next packet in the chain. The constructor `Packet` propagates `'b`.

The type `regularity` is the same as in Fig. 3. The type `chain` exhibits the same structure as the type `chain` in Fig. 3, except that the constructor `Empty` is replaced with the constructor `Ending`, which carries one buffer, and forms a green chain. The type `deque` is defined in the same way as the type `number` in Fig. 3.

We implement the main four operations on deques, as well as a number of auxiliary functions, by translating Kaplan and Tarjan's indications into OCaml code. In this endeavor, we crucially rely on the fact that OCaml's type-checker allows dead branches to be omitted and guarantees complete coverage (that is, verifies that no cases are inadvertently omitted). For example, the function `green_push`, which pushes an element onto a green buffer and

```
type ('a, 'c) buffer =
  | B0 :                            ('a, red           ) buffer (* red *)
  | B1 : 'a                      -> ('a, nogreen * _ * _) buffer (* yellow or red *)
  | B2 : 'a * 'a                 -> ('a, _             ) buffer (* any color *)
  | B3 : 'a * 'a * 'a            -> ('a, _             ) buffer (* any color *)
  | B4 : 'a * 'a * 'a * 'a       -> ('a, nogreen * _ * _) buffer (* yellow or red *)
  | B5 : 'a * 'a * 'a * 'a * 'a -> ('a, red           ) buffer (* red *)

type ('a, 'b, 'c) packet =
  | Hole   :
      ('a, 'a, uncolored) packet
  | Packet :
      ('a, 'c) buffer *                            (* prefix buffer *)
      ('a * 'a, 'b, nogreen * _ * nored) packet * (* child packet *)
      ('a, 'c) buffer ->                           (* suffix buffer *)
      ('a, 'b, 'c) packet

type (_, _) regularity =             (* parameters: packet color and chain color *)
  | G : (green , _ * noyellow * _) regularity
  | Y : (yellow,            green) regularity
  | R : (red   ,            green) regularity

type ('a, 'c) chain =
  | Ending :
      ('a, _) buffer ->
      ('a, green) chain                            (* last level is green *)
  | Chain  :
      ('c1, 'c2) regularity * ('a, 'b, 'c1) packet * ('b, 'c2) chain ->
      ('a, 'c1) chain

type 'a deque =
  | T : ('a, _ * _ * nored) chain -> 'a deque
```

FIGURE 11. Deques (OCaml): types for buffers, packets, chains, and deques

```
let green_push
: type a. a -> (a, green) buffer -> (a, yellow) buffer
= fun x buf ->
  match buf with
  | B2 (a, b)    -> B3 (x, a, b)
  | B3 (a, b, c) -> B4 (x, a, b, c)
```

FIGURE 12. Deques (OCaml): the auxiliary function `green_push`

returns a yellow buffer, is shown in Fig. 12. Because a green buffer must contain two or three elements, only the cases `B2` and `B3` must be considered; all other cases can be omitted.

We write a total of 29 intermediate functions to support the main regularization function, `green_of_red`, which transforms a red chain into a green chain, while preserving the sequence

```
Inductive buffer : Type -> color -> Type :=
  | B0 {A}       :                          buffer A red
  | B1 {A y r}   : A                     -> buffer A (Mix NoGreen y r)
  | B2 {A g y r} : A -> A                -> buffer A (Mix g y r)
  | B3 {A g y r} : A -> A -> A           -> buffer A (Mix g y r)
  | B4 {A y r}   : A -> A -> A -> A      -> buffer A (Mix NoGreen y r)
  | B5 {A}       : A -> A -> A -> A -> A -> buffer A red.

Inductive packet : Type -> Type -> color -> Type :=
  | Hole {A}         :
      packet A A uncolored
  | Packet {A B C y} :
      buffer A C ->
      packet (A * A) B (Mix NoGreen y NoRed) ->
      buffer A C ->
      packet A B C.

Inductive regularity : color -> color -> Type :=
  | G {g r} : regularity green  (Mix g NoYellow r)
  | Y       : regularity yellow green
  | R       : regularity red    green.

Inductive chain : Type -> color -> Type :=
  | Ending {A C}      :
      buffer A C ->
      chain A green
  | Chain {A B C1 C2} :
      regularity C1 C2 -> packet A B C1 -> chain B C2 ->
      chain A C1.

Inductive deque : Type -> Type :=
  | T {A g y} : chain A (Mix g y NoRed) -> deque A.
```

FIGURE 13. Deques (Rocq): types for buffers, packets, chains, and deques

of elements that this chain represents. Once `green_of_red` is available, the four main operations, namely *push*, *pop*, *inject*, and *eject*, are obtained by composing a basic operation, which may produce a red deque or subdeque, with regularization. Our complete OCaml code contains 30 lines of type definitions and 310 lines of code. Here and elsewhere, we count only non-blank, non-comment lines of code, and round up to the nearest ten.

3.3. **Rocq implementation.** Translating these type definitions from OCaml to Rocq is straightforward. The result appears in Fig. 13.

Next, for each data structure, we define a model function. Whereas the model functions defined in Section 2 have result type `nat`, because there the data structures represent natural numbers, the model functions defined in this section have result type `list A`, because the data structures in this section represent sequences of elements. The definitions of these model functions appear in Fig. 14. Their definitions are straightforward. The auxiliary

```
1  Equations buffer_seq {A C} : buffer A C -> list A :=
2  buffer_seq B0       := [];
3  buffer_seq (B1 a)   := [a];
4  buffer_seq (B2 a b) := [a] ++ [b];
5  (* three more cases omitted *).
6
7  Equations flattenp {A} (l : list (A * A)) : list A :=
8  flattenp []            := [];
9  flattenp ((x, y) :: l) := [x] ++ [y] ++ flattenp l.
10
11 Equations packet_seq {A B C} : packet A B C -> list B -> list A :=
12 packet_seq Hole l             := l;
13 packet_seq (Packet p pkt s) l :=
14   buffer_seq p ++ flattenp (packet_seq pkt l) ++ buffer_seq s.
15
16 Equations chain_seq {A C} : chain A C -> list A :=
17 chain_seq (Ending b)       := buffer_seq b;
18 chain_seq (Chain _ pkt cd) := packet_seq pkt (chain_seq cd).
19
20 Equations deque_seq {A} : deque A -> list A :=
21 deque_seq (T c) := chain_seq c.
```

Figure 14. Deques (Rocq): model functions for buffers, packets, and chains

```
Equations green_push {A : Type} (x : A) (b : buffer A green) :
  { b’ : buffer A yellow | buffer_seq b’ = [x] ++ buffer_seq b } :=
green_push x (B2 a b)   := ? B3 x a b;
green_push x (B3 a b c) := ? B4 x a b c.
```

Figure 15. Deques (Rocq): the auxiliary function `green_push`

function `flattenp` flattens a list of pairs of elements into a list of elements. The use of `flattenp` in `packet_seq` is analogous to the use of multiplication by 2 in `packet_nat` (Fig. 8). The parameter `l` in `packet_seq` plays the same role as the parameter `n` in `packet_nat` (Fig. 8): it is the model of the hole. The definition of the model of a non-hole packet (Fig. 14, line 14) indicates that the sequence of elements of a packet is obtained by concatenating the sequence of elements of the prefix buffer, the sequence of elements of the child packet, and the sequence of elements of the suffix buffer, in this order.

As an illustration of the style in which our code is written, Rocq code for the auxiliary function `green_push` is shown in Fig. 15. It is analogous to our OCaml code (Fig. 12), but includes a correctness statement: the postcondition `buffer_seq b’ = [x] ++ buffer_seq b` guarantees that the sequence of elements of the new buffer `b’` is the concatenation of the singleton sequence `[x]` with the sequence of elements of the original buffer `b`.

This creates two proof obligations of the form `buffer_seq b’ = [x] ++ buffer_seq b` where `b’` and `b` are suitably instantiated. In general, our code gives rise to a large number of proof obligations. In every proof obligation, the goal is an equality between two sequences, which we represent in Rocq as lists. In some proof obligations, one or more equalities between sequences are available as hypotheses. Thus, in general, our proof obligations are entailments between equalities on sequences, where sequences are built out of universally quantified variables, the empty list `[]`, singleton lists `[x]`, and list concatenation `++`, and can also be applications of model functions such as `buffer_seq`, `packet_seq`, `chain_seq`, and so on.

Although we do not know of a decision procedure that automatically solves this class of proof obligations, we are able to deal with these proof obligations, in a mostly automated way, thanks to two third-party Rocq modules. First, *Hammer* [Cza20], which provides the tactic `hauto`, is particularly effective. Like `eauto`, it performs goal-directed proof search and can exploit a database of user-provided hints. It is able to automatically discharge many proof obligations. We deal with the remaining proof obligations by hand. Second, to facilitate the remaining manual proofs, we exploit *AAC_tactics* [BP11], which provides tactics for deciding entailment modulo AC and for rewriting modulo AC. This helps us exploit the fact that list concatenation `++` is associative and commutative and admits the empty list `[]` as a unit.

Thanks to these third-party tactics, we are able to verify the correctness of all functions while performing only four manual proofs. Our Rocq file for deques contains 30 lines of type definitions, 40 lines of definitions of model functions, and 390 lines of definitions of operations. The result is a verified implementation of Kaplan and Tarjan's non-catenable deques.

4. Catenable deques: presentation

Kaplan and Tarjan [KT99] design *catenable deques*, a data structure that supports the full set of operations—namely *push*, *pop*, *inject*, *eject*, and *concat*—in worst-case constant time. We refer to catenable deques as *cadeques*, for short.

In this section, we first describe the tree representation of cadeques, without buffers (Section 4.1). This representation involves trees whose nodes carry a color. Based on this representation, we decompose trees into packets and impose regularity constraints. This leads us to the pointer representation of cadeques, still without buffers (Section 4.2). In this representation, packets and chains are primitive concepts, and the regularity invariant holds by construction. We are then ready to describe how buffers are set up and used. As in the previous section, each node is expected to carry a prefix buffer and a suffix buffer. This time, though, a more complex form of recursive slowdown is involved (Section 4.3). We revisit the tree representation of cadeques, this time with buffers (Section 4.4). We give the size constraints that bear on buffers and indicate how the sizes of the buffers determine the assignment of colors to nodes.

The tree representation, with buffers, is a complete description of the data structure. In an implementation, in order to achieve worst-case time complexity $O(1)$, the pointer representation, with buffers, must be used instead. We do not explicitly describe this representation in this section, because it is visible in our OCaml implementation of cadeques, which is the subject of Section 5.

Implementing cadeques in Rocq turns out to be somewhat challenging. A discussion of the issues and a presentation of our Rocq implementation appear in Sections 6 to 8.

4.1. Tree representation, without buffers.

Whereas the skeleton of a non-catenable deque has linear structure, the skeleton of a catenable deque has nontrivial tree structure. Briefly put, it is a forest of trees of colored nodes, where each node carries zero, one, or two

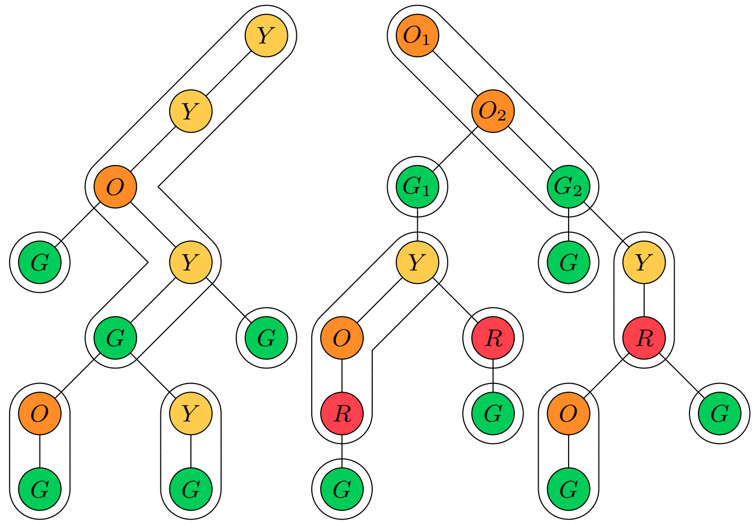


FIGURE 16. Cadeques: abstract structure and decomposition into packets

subtrees. Because of the move from a setting where a node has at most one successor to a setting where a node has at most two successors, a new color, orange, must be introduced.

- A *color* is now one of red, orange, yellow, and green.

4.1.1. *Trees and forests.* The trees of interest are given by two mutually inductive definitions:

- A *tree* consists of a color and a forest.
- A *forest* consists of zero, one, or two trees.

This can be visually understood as follows: a tree consists of a root node, which carries a color, and a number of subtrees, which together form a forest. The number of subtrees, which is known as the *arity* of the root node, must be 0, 1, or 2.

On top of this definition, one well-formedness condition is imposed:

- A node of arity 0 must be green.

4.1.2. *Decomposing trees and forests into packets.* A well-formed tree or forest can be decomposed into a collection of disjoint packets, where a *packet* is a downward path that traverses a (possibly null) number of orange and yellow nodes and ends at a green or red node. This decomposition exists and is unique; it is determined by the colors and arities of the nodes. It is described by the following rules:

- A packet that reaches a green or red node ends at this node.
  Each subtree is independently decomposed into packets.
- A packet that reaches an orange or yellow node of arity 1
  extends down to this node's single child.
- A packet that reaches an orange or yellow node of arity 2
  extends down to this node's *preferred child*.
  - An orange node of arity 2 prefers its right-hand child.
  - A yellow node of arity 2 prefers its left-hand child.

  In either case, the non-preferred subtree is independently decomposed into packets.

(RC1) Trivial.
(RC2) A packet that begins at the left child of an orange node must be green.
(RC3) A packet that begins at a child of a red node must be green.
(RC4) The top packets must be green.

FIGURE 17. Cadeques: regularity constraints

In Kaplan and Tarjan's paper, our packets are known as *preferred paths*. A packet goes through zero or more orange or yellow nodes (whose arity is 1 or 2) and ends at a green or red node. We refer to the sequence of orange or yellow nodes as the *body* of the packet; we refer to the final green or red node as the *tail* of the packet.

In the previous sections (Section 2, Section 3), a packet *begins* with a distinguished green or red node, followed with a number of yellow nodes. In contrast, in this section, a packet *ends* with a distinguished green or red node, preceded by a number of orange or yellow nodes. This design seems forced upon us by the following two facts: (a) packets have linear structure, that is, they are paths; (b) a green or red node can have arity 2, and does not have a preferred child.

An example of a forest, and its decomposition into disjoint packets, are shown in Fig. 16. A node is represented by a colored disc. A child relationship between two nodes is represented by a straight edge. A packet is represented by a curved line that surrounds a group of nodes. The forest in Fig. 16 is composed of two trees. Let us explain the structure of the packet that begins at the root of the second tree. This packet begins at an orange node of arity 1, named O_1. It extends down to its child, O_2, an orange node of arity 2. It then extends down to the preferred child of node O_2, the green node G_2, and ends there. The non-preferred child of node O_2, the green node G_1, forms the beginning of a new packet. Because G_1 is a green node, this new packet ends there; the body of this packet is empty. The children of the green nodes G_1 and G_2 form the beginning of new packets; and so on.

4.1.3. *Regularity.* We now assign colors to packets, as follows:

- The color of a packet is the color of its tail node.

Because a tail node is green or red, the color of a packet is green or red.

Furthermore, we impose constraints, shown in Fig. 17, on the coloring of packets. We refer to constraints (RC2) and (RC3) as the *semi-regularity* constraints. We refer to the constraints (RC2), (RC3), and (RC4) together as the *regularity* constraints. These constraints can be loosely compared with the earlier regularity constraints of Fig. 1, which concern the redundant binary representation of natural numbers, as follows. Because every packet is green or red, constraint (R1) becomes the vacuously true constraint (RC1). Constraint (R2) corresponds loosely to constraint (RC2). Constraint (R3) becomes constraint (RC3). Constraint (R4) becomes constraint (RC4).

4.2. **Pointer representation, without buffers.** In the tree representation, trees and forests have extremely simple structure (Section 4.1.1). *A posteriori*, they can be decomposed into packets (Section 4.1.2) and subjected to regularity constraints (Section 4.1.3). In the pointer representation, in contrast, the decomposition into packets and the regularity constraints are imposed *a priori*.

4.2.1. *Packets and chains.* In the pointer representation, instead of working at the granularity of individual nodes, one works at the granularity of packets. Furthermore, whereas the tree representation is based on trees and forests, the pointer representation is based on *chains* whose arity can be 0, 1, or 2.

One can think of the arity of a chain as the number of trees that this chain represents. A chain of arity 1, which has a root packet and a child chain, corresponds to a tree. A chain of arity 0, 1, or 2 corresponds to a forest of 0, 1, or 2 trees.

In the following, we also speak of the arity of a node (or packet). One can think of it as the number of children that this node (or packet) expects. In other words, one can think of it as a constraint on the arity of the child chain that follows this node (or packet).

The pointer representation is defined as follows. We give a definition of *nodes*, followed with mutually inductive definitions of *chains of arity* a, *packets of arity* a, and *bodies*.

- A *node* is a pair of a color and an arity, where an arity a is a member of $\{0, 1, 2\}$.
  A node's arity can be 0 only if its color is green.
- A *chain* of arity a is defined as follows:
  - A chain of arity 0 is empty.
  - A chain of arity 1 consists of a packet of arity a' (the *root packet*)
    and a chain of arity a' (the *child chain*), for some a'.
    If the root packet is red then the child chain must be green.
    (*A chain is green* if its zero, one, or two root packets are green.)
  - A chain of arity 2 consists of two chains of arity 1.
- A *packet* of arity a consists of a body and a green or red node of arity a (the *tail node*).
  The color of a packet is the color of its tail node.
- A *body* is one of the following:
  - Empty.
  - An orange or yellow node of arity 1
    and a body (the *continuation*).
  - An orange node of arity 2,
    a chain of arity 1 whose root packet is green,
    and a body (the *continuation*).
  - A yellow node of arity 2,
    a body (the *continuation*),
    and a chain of arity 1.

This representation allows moving in constant time "from a packet to its children", that is, from a chain of arity 1 to its child chain. It also allows moving in constant time from a packet to its tail node.

4.2.2. *Correspondence between tree representation and pointer representation.* The tree representation (Section 4.1) and the pointer representation (above) are isomorphic: they are alternative representations of the same data. Indeed, a chain, as defined above, can be decoded into a well-formed forest (Section 4.1.1), which, once decomposed into packets as per the rules of Section 4.1.2, satisfies the semi-regularity constraints of Section 4.1.3. If this chain is green, then the resulting forest also satisfies the regularity constraint of Section 4.1.3. Conversely, every forest that is well-formed and satisfies the semi-regularity constraints can be encoded as a chain, as defined above.

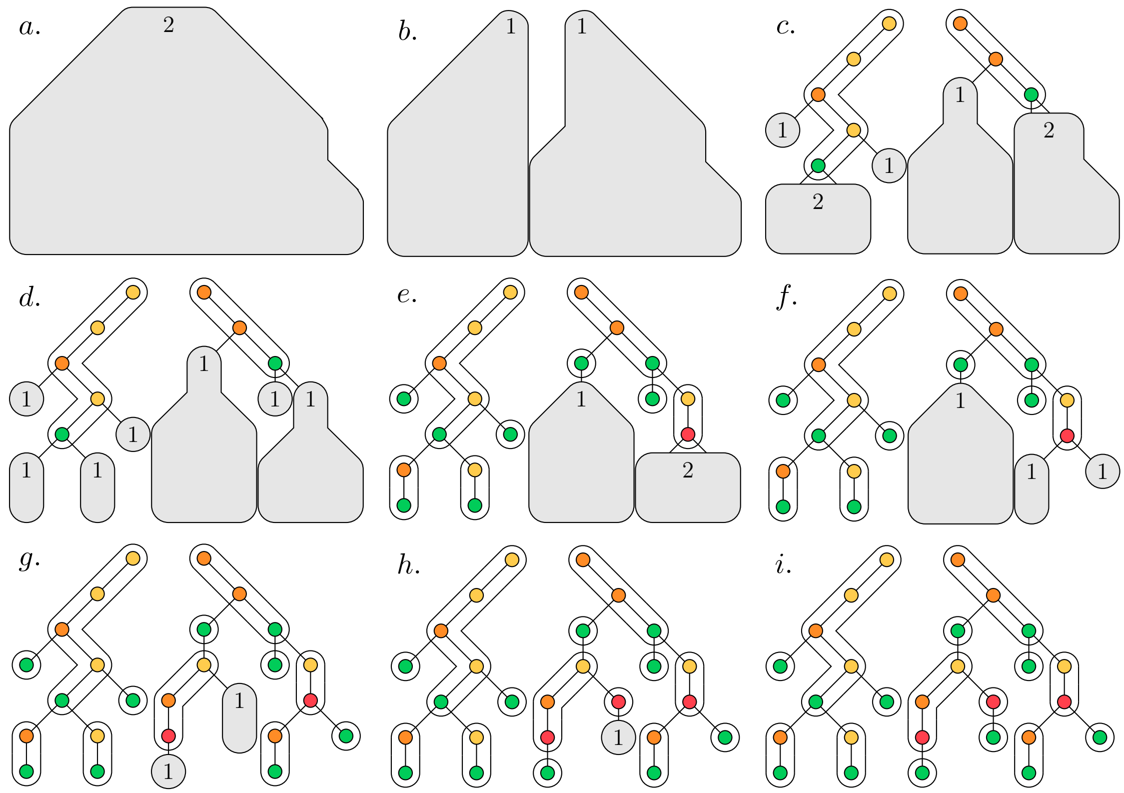


FIGURE 18. Gradually unfolding a chain into a forest (decomposed into packets)

The process of gradually decoding a chain into a forest is illustrated in Fig. 18. A chain that has not yet been decoded is depicted as an opaque area. Its arity is indicated by the number 1 or 2 inside this area. (Chains of arity 0 are not shown, as they are empty.) At stage (a), a chain of arity 2 stands for the entire forest shown in Fig. 16. At stage (b), this chain of arity 2 has been unfolded into two chains of arity 1. At stage (c), each of these chains of arity 1 has been unfolded into a root packet and a child chain. Furthermore, the root packet itself has been unfolded into a body and a tail node, and the body has been unfolded into a sequence of orange or yellow nodes, some of which carry chains. In the following stages, the process continues until the entire forest (and its decomposition into packets) emerge.

The remarks in the previous paragraphs are not essential. They are provided only in the hope that they may help the reader understand the mutually inductive structure of chains, packets, and bodies.

4.3. **Buffers and recursive slowdown.** The structure that we have just described is abstract, or incomplete: it is just a skeleton. To obtain a complete description of cadeques, one must enrich this structure with elements and buffers. Both the tree representation and the pointer representation can be enriched in this way.

The introduction of buffers can be briefly and partially described as follows.

(1) Every data structure is parameterized with a type A, the type of its elements.
(2) Every node carries a *prefix buffer* and a *suffix buffer* that contain elements of type A.
(3) A buffer is a non-catenable deque, as defined in Section 3.
(4) As one goes from a node down to a child, the parameter A is changed to "stored triple of A", where "stored triple" is one of the data types involved in the definition of cadeques.

In comparison with Section 3, the nature of buffers changes. There, a buffer was a tuple of bounded size. Here, it is a non-catenable deque, whose size is unbounded. (By the *size* of a data structure, we mean the number of elements that it contains.)

Recursive slowdown is still present, but follows a more complex and more deeply recursive pattern than in Section 3. There, as one went from a node down to its child, the parameter A was transformed to $A \times A$. Here, it is changed to "stored triple of A", where "stored triple" is one of the data types that we are in the process of defining. This makes cadeques a *truly nested data type* [AMU05].

To complete the description of buffers, a connection between buffer sizes and colors must be established. Very roughly, buffer sizes must be subjected to certain constraints, and the color of a node must be related with the sizes of the buffers that it carries. This is spelled out in the next subsection.

4.4. **Tree representation, with buffers.** To describe buffers in full detail, it is preferable to do so in the setting of the tree representation first. This description can later be adapted to the setting of the pointer representation.

4.4.1. *Triples and cadeques.* The tree representation of cadeques involves two mutually recursive types, namely trees and forests (Section 4.1.1). Once this representation is enriched with buffers, it is still described by two mutually recursive types, namely triples and cadeques. A triple is an enriched tree; a cadeque is an enriched forest. Here is a definition of these types [KT99, §6.1]:

- A *triple* of elements of type A is composed of:
  - a deque, also known as a *prefix buffer*, whose elements have type A;
  - a cadeque, also known as the *child cadeque*, whose elements are triples of elements of type A, which Kaplan and Tarjan refer to as *stored triples*; and
  - a deque, also known as a *suffix buffer*, whose elements have type A.
- A *cadeque* is either:
  - empty;
  - composed of a single triple, which Kaplan and Tarjan refer to as an *only triple*; or
  - composed of two triples, which they refer to as a *left triple* and a *right triple*.

The four kinds of triples mentioned above can be divided in two independent categories. On the one hand, stored triples exist *inside* buffers: they serve as elements of buffers. When buffers are abstracted away, they disappear altogether. On the other hand, only triples, left triples, and right triples exist *outside* buffers. When buffers are abstracted away, they form the skeleton that remains. We refer to them collectively as *ordinary triples*. In our OCaml and Rocq implementations, stored triples and ordinary triples form two distinct types.

4.4.2. *Kinds.* To distinguish between the three varieties of ordinary triples, we say that the *kind* of an ordinary triple is "only", "left", or "right".

Above, we have indicated that an ordinary triple corresponds to a subtree in the skeleton. The kind of an ordinary triple records the relationship between this subtree and its parent: it indicates whether this subtree is the only child, the left child, or the right child of its parent.

4.4.3. *Size constraints.* Kaplan and Tarjan impose size constraints on buffers. In the case of stored triples, the constraints are as follows:

(ST1) In a stored triple, both buffers must have at least 3 elements,
(ST2) unless the child cadeque and one of the buffers are empty,
in which case the other buffer must have at least 3 elements.

In the case of ordinary triples, the constraints are as follows:

(OT1) In an only triple, both buffers must have at least 5 elements,
(OT2) unless the child cadeque and one of the buffers are empty,
in which case the other buffer must contain at least 1 element.
(OT3) In a left triple, the prefix buffer must have at least 5 elements,
while the suffix buffer must have exactly 2 elements.
(OT4) In a right triple, the prefix buffer must have exactly 2 elements,
while the suffix buffer must have at least 5 elements.

4.4.4. *Color assignment.* Kaplan and Tarjan define a mapping of sizes to colors:

(C5) size 5 is red;
(C6) size 6 is orange;
(C7) size 7 is yellow;
(C8) size 8 and greater sizes are green.

This mapping allows assigning a color to a buffer, based on the size of this buffer, provided this size is at least 5. Then, Kaplan and Tarjan assign colors to ordinary triples, based on their kind and on the colors of some of their buffers:

(BC1) An ordinary triple whose child cadeque is empty is green.
(BC2) An ordinary triple whose child cadeque is nonempty obeys the following rules:
- (BC2a) The color of an only triple is the worst color of its two buffers.
- (BC2b) The color of a left triple is the color of its prefix buffer.
- (BC2c) The color of a right triple is the color of its suffix buffer.

The ordering of colors, from worst to best, is red < orange < yellow < green.

4.4.5. *Regularity.* As each ordinary triple now implicitly carries a color, one can decompose triples and cadeques into packets, and impose regularity constraints, in the same manner as earlier (Sections 4.1.2 and 4.1.3).

4.5. **Pointer representation, with buffers.** Of four ways of looking at Kaplan and Tarjan's cadeques, we have just presented three. Indeed, we have presented a skeletal structure, without buffers, under two equivalent representations, namely the tree representation (Section 4.1) and the pointer representation (Section 4.2). Then, we have introduced buffers and have given a complete description of cadeques under the tree representation (Section 4.4). At this point, we are ready to describe the fourth and last view, that is, the pointer representation of cadeques, with buffers. We do not give an English definition of it. Instead, we move directly to our OCaml definition.

5. Catenable deques: OCaml implementation

We now transcribe the ideas of the previous section into OCaml type definitions and code.

5.1. **Buffers: size-indexed deques.** As explained earlier (Section 4.3), within a catenable deque, a *buffer* is a non-catenable deque, subject to certain size constraints. Thus, our implementation of deques (Section 3.2) can be used as a building block in our implementation of cadeques.

However, as far as static type-checking is concerned, the API offered by our deques is not quite as expressive as we would like. This API consists of a parameterized type `'a deque` (Fig. 11), which is equipped with a number of constants and operations, such as `empty`, `push`, `pop`, `inject`, and `eject`. It offers no way of referring at the type level to the size of a deque.[10] Therefore, if we chose to build our implementation of cadeques directly on top of this API, we would be unable to express at the type level the size constraints of Section 4.4.3. Thus, the OCaml type-checker would be unable to statically verify that our implementation of cadeques respects these size constraints. Yet, such a static check is highly desirable: it is a crucial help, and a guide, while developing the code.

To address this problem, on top of the deque API of Section 3.2, we develop a richer API where the type of deques is indexed with a type-level natural number `'n`, which represents a lower bound[11] on the size of the deque. This API lets us express the constraints that impose a constant lower bound on the size of a buffer, that is, all constraints of the form: "this buffer must have at least k elements", where k is a constant.[12]

This richer API and its construction appear in Figs. 19 and 20. The construction relies on type abstraction. Outside of the abstraction barrier (that is, from a user's point of view), the type of buffers, `('a, 'n) Buffer.t`, is an abstract type, whose parameter `'n` represents a lower bound on the size of the buffer. Inside the abstraction barrier, the parameter `'n` is irrelevant: indeed, the type `('a, 'n) t` is defined as a synonym for `'a deque` (line 10 in Fig. 20). This yields a form of one-sided security. Inside the abstraction barrier, runtime assertions (lines 15 and 18) ensure that the parameter `'n` is indeed a lower bound on the size

[10] This API could offer a `size` function, which computes and returns the size of a deque. However, an expression of the form `size(d)` cannot appear within a type: OCaml does not have dependent types.

[11] It would seem more natural for this index to represent the size of the deque, as opposed to a lower bound on the size of the deque. However, some operations in the deque API, such as `push_vector` (not shown in Fig. 20), insert an undetermined number of elements into a deque. For this reason, only a lower bound on the size of the deque can be statically known.

[12] This API does not let us express the constraint: "this buffer must have exactly two elements". Fortunately, we are able to remove the need for this constraint: where a buffer of exactly two elements is required, we instead use two naked elements. This is visible in the definition of the type `node` (Section 5.2.3).

```
type    z                       (* The number 0   *)
type 'n s                       (* The number n+1 *)

type      eq0 = z               (* The number 0   *)
type      eq1 = z s             (* The number 1   *)
type      eq2 = z s s           (* The number 2   *)

type 'n plus1 = 'n s            (* The number n+1 *)
type 'n plus2 = 'n s plus1      (* The number n+2 *)
type 'n plus3 = 'n s plus2      (* The number n+3 *)
type 'n plus4 = 'n s plus3      (* The number n+4 *)
type 'n plus5 = 'n s plus4      (* The number n+5 *)
```

FIGURE 19. Deques (OCaml): type-level natural numbers

```
module Buffer : sig
  type ('a, 'n) t
  val empty : ('a, z) t
  val push   : 'a -> ('a, 'n) t -> ('a, 'n s) t
  val inject : ('a, 'n) t -> 'a -> ('a, 'n s) t
  val pop    : ('a, 'n s) t -> 'a * ('a, 'n) t
  val eject : ('a, 'n s) t -> ('a, 'n) t * 'a
  (* Additional functions omitted *)
end = struct
  type ('a, 'n) t = 'a Deque.t
  let empty  = Deque.empty
  let push   = Deque.push
  let inject = Deque.inject
  let pop t = match Deque.pop t with
    | None -> assert false (* this cannot happen *)
    | Some (x, t') -> (x, t')
  let eject t = match Deque.eject t with
    | None -> assert false (* this cannot happen *)
    | Some (t', x) -> (t', x)
  (* Additional functions omitted *)
end
```

FIGURE 20. Cadeques (OCaml): size-indexed deques, also known as buffers

of the buffer. Outside the abstraction barrier, this parameter is meaningful and can be used to express size constraints, which the OCaml type-checker statically enforces. This idiom is known in the literature as a phantom type parameter [LM99].

The operations `push` and `inject` increase the size of a buffer by one. Therefore, a lower bound on the size of the buffer can be safely increased by one, as well. The types assigned to `push` and `inject` (lines 4 and 5 in Fig. 20) reflect this idea.[13] The operations `pop` and `eject` must be applied only to a nonempty buffer, and decrease the size of this buffer by one. Therefore, a lower bound on the size of the buffer must be decreased by one. The types assigned to `pop` and `eject` (lines 6 and 7) reflect these two requirements. The fact that `pop` and `eject` can be applied only to a nonempty buffer guarantees that the runtime assertions at lines 15 and 18 cannot fail; however, this informal argument is not machine-checked.

5.2. **Nodes.** We now wish to define a type of *nodes*. At a concrete level, we want a node to be an ordinary triple, as defined in Section 4.4.1, deprived of its child cadeque. Thus, roughly speaking, a node should be a pair of buffers. Furthermore, a node should be subject to the size constraints that bear on ordinary triples. Four cases, (OT1) to (OT4), have been listed in Section 4.4.3. Every node must match exactly one of these cases. Thus, nodes should inhabit a sum type whose four constructors correspond to these four cases.

5.2.1. *Type-level properties of nodes.* We need to keep track, at the type level, of four properties of a node, namely the type of its elements, its arity, its kind, and its color. An arity is 0, 1, or 2 (Section 4.2.1). A *kind* is "only", "left", or "right" (Section 4.4.2).[14] Therefore, we parameterize the type `node` with four parameters: this type takes the form `('a, 'arity, 'kind, 'c) node`.

The parameter `'a` is the type of the elements of this node.

The parameter `'arity` is meant to range over type-level arities.[15] A type-level arity is a type-level natural number among 0, 1, and 2. We introduce `empty`, `single`, and `pair` as convenient synonyms for these three type-level natural numbers (Fig. 21).

The parameter `'kind` is meant to range over type-level kinds. We introduce `only`, `left`, and `right` as three distinct type-level kinds (Fig. 21).

The parameter `'c` is meant to range over type-level colors, whose definition appears in Fig. 22. This definition obeys the same scheme as our earlier definition (Fig. 2). This time, a new orange hue is added; colors are now composed of four hue bits.

5.2.2. *Color assignment.* To express the coloring rules of Section 4.4.4, we define the type `('prefix_delta, 'suffix_delta, 'arity, 'c) node_coloring`. This type encodes a relation between the sizes of a node's prefix and suffix buffers, its arity, and its color. An inhabitant of this type is a witness that the relation holds.

The type variables `'prefix_delta` and `'suffix_delta` are meant to range over type-level natural numbers. In the case of an only node, where the sizes of both buffers determine the node's color (BC2a), we intend to instantiate `'prefix_delta` and `'suffix_delta` respectively with the size of the prefix buffer minus 5 and with the size of the suffix buffer minus 5, thereby constraining the sizes of both buffers. In the case of a left node, where only the size of the prefix buffer matters (BC2b), we intend to instantiate `'prefix_delta` and `'suffix_delta` respectively with the size of the prefix buffer minus 5 and with a dummy

[13] It would be safe for `push` to have result type `('a, 'n) t` instead of `('a, 'n s) t`. This would encode a weaker lower bound, namely n, instead of $n+1$. Some operations not shown in Fig. 20, such as `push_vector`, introduce an under-approximation in this way.

[14] A node whose kind is "only" is an *only node*. Similarly, a node whose kind is "left" is a *left node*, and a node whose kind is "right" is a *right node*.

[15] This cannot be explicitly expressed in OCaml.

```
(* An arity is 0, 1, or 2. *)
type empty  = eq0
type single = eq1
type pair   = eq2

(* A kind is only, left, or right. *)
type only
type left
type right
```

FIGURE 21. Cadeques (OCaml): type-level arities and kinds

```
(* A group of distinct types encode the presence or absence of a hue. *)
type somegreen  = SOME_GREEN
type nogreen    = NO_GREEN
type someyellow = SOME_YELLOW
type noyellow   = NO_YELLOW
type someorange = SOME_ORANGE
type noorange   = NO_ORANGE
type somered    = SOME_RED
type nored      = NO_RED
(* A color is a quadruple of four hue bits. *)
type green      =  somegreen *   noyellow *   noorange *   nored
type yellow     =    nogreen * someyellow *   noorange *   nored
type orange     =    nogreen *   noyellow * someorange *   nored
type red        =    nogreen *   noyellow *   noorange * somered
```

FIGURE 22. Cadeques (OCaml): type-level colors

variable, thereby constraining only the size of the prefix buffer. Symmetrically, in the case of a right node (BC2c), we intend to constrain only the size of the suffix buffer. This will be visible in Section 5.2.3, where the `node_coloring` relation is used in the definition of the type `node`.

The definition of `node_coloring` is shown in Fig. 23. The constructor `EN` concerns a node of arity 0; the other four constructors concern a node of nonzero arity.

"A node of arity 0" can also be understood as "a node whose child cadeque has arity 0", that is, "a node whose child cadeque is empty". Therefore, the constructor `EN`, which states that a node of arity 0 is green, corresponds to (BC1).

Regarding nodes of nonzero arity, the constructors `GN`, `YN`, `ON`, and `RN` encode four implications: if the node's color is green then the two "deltas" must be at least 3; if its color is yellow then they must be at least 2; if it is orange then they must be at least 1; and if it is red then fine. This encodes Rules (C5)–(C8). In fact, this encodes a relaxed version of these rules: as in the case of non-catenable deques (Section 3.2), we allow a node to receive a suboptimal color.

5.2.3. *Nodes.* The definition of the type `node` appears in Fig. 23. As announced earlier, this is a sum type whose four constructors correspond to (OT1)–(OT4).

```
(* Convenient synonyms. *)
type ('a, 'n) prefix = ('a, 'n) Buffer.t
type ('a, 'n) suffix = ('a, 'n) Buffer.t

(* Constraints on buffer sizes, arity, and color. *)
(* Delta is size minus 5. *)
type ('prefix_delta, 'suffix_delta, 'arity, 'c) node_coloring =
  | EN : (       _,        _,     eq0, green ) node_coloring
  | GN : (_ plus3, _ plus3, _ plus1, green ) node_coloring
  | YN : (_ plus2, _ plus2, _ plus1, yellow) node_coloring
  | ON : (_ plus1, _ plus1, _ plus1, orange) node_coloring
  | RN : (       _,        _, _ plus1, red   ) node_coloring

(* Nodes. *)
type ('a, 'arity, 'kind, 'c) node =
  | Only      : ('pdelta, 'sdelta, 'n plus1, 'c) node_coloring
              * ('a, 'pdelta plus5) prefix
              * ('a, 'sdelta plus5) suffix
             -> ('a, 'n plus1, only, 'c) node
  | Only_end  : ('a, _ plus1) Buffer.t
             -> ('a, eq0, only, green) node
  | Left      : ('pdelta, _, 'arity, 'c) node_coloring
              * ('a, 'pdelta plus5) prefix
              * ('a * 'a)
             -> ('a, 'arity, left, 'c) node
  | Right     : (_, 'sdelta, 'arity, 'c) node_coloring
              * ('a * 'a)
              * ('a, 'sdelta plus5) suffix
             -> ('a, 'arity, right, 'c) node
```

FIGURE 23. Cadeques (OCaml): types for nodes

The constructors `Only` and `Only_end` construct an only node, that is, a node whose kind is `only`. The constructor `Left` constructs a left node, whose kind is `left`. The constructor `Right` constructs a right node, whose kind is `right`.

- The constructor `Only` corresponds to (OT1). It constructs a node of nonzero arity. It carries two buffers. The sizes of these buffers as well as the node's arity and color are subject to a coloring constraint.
- The constructor `Only_end` corresponds to (OT2). It constructs a node of arity 0. It carries only one buffer, which must contain at least one element. Its color is green. No coloring constraint is needed in this case.
- The constructor `Left` corresponds to (OT3). It carries a prefix buffer of size at least 5 and a suffix buffer of size 2. The size of the prefix buffer as well as the node's arity and color are subject to a coloring constraint. The suffix buffer is represented simply as a pair of elements.
- The constructor `Right` corresponds to (OT4). It is symmetric with `Left`.

```
type 'a stored =
| Big :
      ('a, _ plus3) prefix                      (* prefix buffer *)
    * ('a stored, _, only, _, _) chain          (* child chain *)
    * ('a, _ plus3) suffix                      (* suffix buffer *)
   -> 'a stored
| Small :
      ('a, _ plus3) Buffer.t                    (* just one buffer *)
   -> 'a stored

and ('a, 'b, 'hk, 'tk) body =
| Hole :
      ('a, 'a, 'k, 'k) body
| Single_child :
      ('a, eq1, 'hk, nogreen*_*_*nored) node    (* orange or yellow node of arity 1 *)
    * ('a stored, 'b, only, 'tk) body               (* child: continuation of body *)
   -> ('a, 'b, 'hk, 'tk) body
| Pair_orange :
      ('a, eq2, 'hk, orange) node               (* orange node of arity 2 *)
    * ('a stored, single, left, green, green) chain (* left: green chain of arity 1 *)
    * ('a stored, 'b, right, 'tk) body              (* right: continuation of body *)
   -> ('a, 'b, 'hk, 'tk) body
| Pair_yellow :
      ('a, eq2, 'hk, yellow) node               (* yellow node of arity 2 *)
    * ('a stored, 'b, left, 'tk) body               (* left: continuation of body *)
    * ('a stored, single, right, 'c, 'c) chain      (* right: a chain of arity 1 *)
   -> ('a, 'b, 'hk, 'tk) body

and ('a, 'b, 'arity, 'kind, 'c) packet =
| Packet :
      ('a, 'b, 'kind, 'tk) body                 (* body *)
    * ('b, 'arity, 'tk, _*noyellow*noorange*_ as 'c) node (* green or red tail node *)
   -> ('a, 'b stored, 'arity, 'kind, 'c) packet

and ('a, 'arity, 'kind, 'lc, 'rc) chain =
| Empty :
      ('a, empty, _, _, _) chain
| Single :
      ('c, 'lc1, 'rc1) regularity               (* regularity constraint *)
    * ('a, 'b, 'arity1, 'kind, 'c) packet       (* root packet *)
    * ('b , 'arity1, only, 'lc1, 'rc1) chain    (* child chain *)
   -> ('a, single, 'kind, 'c, 'c) chain
| Pair :
      ('a, single, left , 'lc, 'lc) chain       (* left single chain *)
    * ('a, single, right, 'rc, 'rc) chain       (* right single chain *)
   -> ('a, pair, _, 'lc, 'rc) chain
```

Figure 24. Cadeques (OCaml): types for stored triples, bodies, packets, and chains

5.3. **Chains and packets.** We now present the definitions of the types `stored`, `body`, `packet`, and `chain`. These four definitions are mutually recursive. They appear in Fig. 24. In the following, we first explain the five parameters of the type `chain`. Then, we describe each of these four definitions in turn. Finally, we present the type of cadeques.

5.3.1. *Type-level properties of chains.* We keep track, at the type level, of five properties of a chain, namely the type of its elements, its arity, the kind of its root node (when it has a single root node), and two colors that describe its root node or root nodes. Thus, we parameterize the type `chain` with five parameters: this type takes the form `('a, 'arity, 'kind, 'lc, 'rc) chain`.

The parameter `'a` is the type of the elements of this chain.

The parameter `'arity` ranges over type-level arities. In Section 5.2.1, we have introduced `empty`, `single`, and `pair` as synonyms for the type-level numbers 0, 1, and 2. The arity of a chain indicates how many trees this chain contains, when it is viewed as a forest.

The parameter `'kind` represents the kind of the chain's root node. This parameter is meaningful only in the case of single chains (that is, chains of arity 1), which have a single root node. In the cases of empty chains and pair chains, it is unconstrained.

The parameters `'lc` and `'rc` represent the color or colors of this chain. The names `'lc` and `'rc` stand for "left color" and "right color". In the case of an empty chain, these parameters are unconstrained. In the case a single chain, they both represent the color of the root packet of the chain. In the case of a pair chain, they represent the colors of the two root packets.

5.3.2. *Stored triples.* The definition of the type `'a stored` appears in Fig. 24. A stored triple is either *big* or *small*: this corresponds to the cases (ST1) and (ST2) in Section 4.4.3.

A big stored triple consists of a prefix buffer, a child chain, and a suffix buffer, where each buffer contains at least three elements. As indicated in Section 4.4.1, recursive slowdown takes place at this point: in a big stored triple whose elements have type `'a`, the child chain contains elements of type `'a stored`.

A small stored triple consists of a single buffer that contains at least three elements.

5.3.3. *Bodies.* Next in Fig. 24 comes the definition of the type `('a, 'b, 'hk, 'tk) body`. From Section 4.2.1, recall that a body and a tail node, together, form a packet. One can think that the tail node is "placed in the hole" at the end of the body.

The parameter `'a` is the type of the elements of this body. The parameter `'b` is the type of the elements of the tail node that follows this body. These parameters play the same role as the parameters by the same names in the type `('a, 'b, 'c) packet` of non-catenable deques (Fig. 11).

The parameter `'hk`, for "head kind", is the kind of the head node of this body, if there is such a node, that is, if this body is nonempty. Otherwise, it is the kind of the tail node that follows this body. The parameter `'tk`, for "tail kind", is the kind of the tail node that follows this body.

The four constructors correspond to the four cases in the description of bodies that was given in Section 4.2.1.

- The constructor `Hole` represents an empty body. In this case, because the "head node" is the tail node, the parameters `'a` and `'b` are equated, and the parameters `'hk` and `'tk` are equated.

- The constructor `Single_child` carries a root node of arity 1, which must be orange or yellow, and a child body, which represents the continuation of the parent body. Because the child body is the only child of its parent, its head kind must be `only`. As one moves from parent to child, recursive slowdown takes place: whereas the parent body has elements of type `'a`, the child body has elements of type `'a stored`.
- The constructor `Pair_orange` carries a root node of arity 2, which must be orange, and two children. The left child, which is the non-preferred child, is a green chain of arity 1; the right child, which is the preferred child, is a body, which represents the continuation of the parent body. Because the child chain is the left child of its parent, its kind must be `left`. Because the child body is the right child of its parent, its head kind must be `right`. As one moves from parent to child, recursive slowdown takes place: both children have elements of type `'a stored`.
- The constructor `Pair_yellow` is symmetric but places no color constraint on the child chain.

5.3.4. *Packets.* The definition of the type `('a, 'b, 'arity, 'kind, 'c) packet` is shown in Fig. 24.

The parameter `'a` is the type of the elements of this packet. The parameter `'b` is the type of the elements of the chain that follows this packet. The parameter `'arity` is the arity of this packet. By definition (Section 4.2.1), it is the arity of the tail node of this packet. It is also the arity of the chain that follows this packet. The parameter `'kind` is the kind of the root node of this packet. The parameter `'c` is the color of this packet. By definition (Section 4.2.1), it is the color of the tail node of this packet.

A packet is a pair of a body and a tail node, which must be green or red. The pattern `_*noyellow*noorange*_` encodes "green or red". Writing `_*noyellow*noorange*_ as 'c` allows us to simultaneously name this color `'c` and require it to match this pattern.

5.3.5. *Chains.* The definition of the type `chain` is the last one in Fig. 24. There are three constructors, which correspond to chains of arity 0, 1, and 2, as announced in Section 4.2.1.

- The constructor `Empty` constructs a chain of arity 0, that is, an *empty chain*.
- The constructor `Single` constructs a chain of arity 1, that is, a *single chain*. It carries a root packet and a child chain. The kind `'kind` and color `'c` of the chain are the kind and color of its root packet. The arity `'arity1` and the element type `'b` of the child chain are dictated by the root packet. A regularity constraint relates the color of the root packet and the colors of the root packets of the child chain. As indicated in Section 4.2.1, this regularity constraint states that if the root packet is red then the following packets must be green. This corresponds to constraint (RC3) in Fig. 17. A detailed commentary of the type `regularity` is deferred to Section 5.3.6.
- The constructor `Pair` carries two chains of arity 1 and constructs a chain of arity 2, that is, a *pair chain*. The left child chain has kind `left`; the right child chain has kind `right`. The colors `'lc` and `'rc` respectively describe the left child chain and the right child chain.

5.3.6. *Regularity.* The color of a packet must be green or red (Section 4.1.3). Therefore, in the definition of the data constructor `Single` (Fig. 24, line 39), the color `'c` must be green or red. In view of this remark, it is evident that merely two constructors suffice for the type `regularity`, namely `G` and `R` (Fig. 25). Translated into informal English, the types of these constructors can be read as follows:

```
(* A relation between the color 'c of a packet and the
   color(s) 'lc and 'rc of the following packet(s). *)
type ('c, 'lc, 'rc) regularity =
  | G : (green,     _,     _) regularity
  | R : (  red, green, green) regularity
```

FIGURE 25. Cadeques (OCaml): regularity constraints

```
type 'a cadeque =
  T :
     ('a, _, only, green, green) chain
  -> 'a cadeque
```

FIGURE 26. Cadeques (OCaml): the type of catenable deques

- if the root packet is green, the following packets can have any color;
- if the root packet is red, the following packets must be green.

5.3.7. *Cadeques.* Finally, the type of cadeques is defined in Fig. 26. For a chain to form a stand-alone cadeque, this chain must be green, and its kind must be `only`. The first requirement reflects the regularity constraint (RC4) (Fig. 17). The second requirement admits chains of arity 0, chains of arity 1 whose root node has kind `only`, and chains of arity 2. It rules out "left single chains" and "right single chains", that is, chains of arity 1 whose root node has kind `left` or `right`. Indeed, these are not meant to exist in isolation.

This completes our presentation of the data structure: at this point, all of the type definitions that describe the structure of cadeques have been presented. There remains to briefly describe our implementation of the operations.

5.4. **Operations.** After defining several auxiliary data types and functions, we implement *push*, *pop*, *inject*, *eject*, and *concat*. The definition of buffers (that is, size-indexed deques) occupies 190 lines of code. The auxiliary data types occupy 90 lines. The auxiliary functions and the main five operations occupy 570 lines.

6. OBSTACLES AND WORKAROUNDS

We would now like to port our OCaml definition of catenable deques, which comprises OCaml type definitions and OCaml code, to Rocq. Because we have used only the purely functional subset of OCaml, one might expect a literal translation from OCaml to Rocq to be possible. Unfortunately, this is not the case. Because Rocq's type system is designed to enforce termination, it is in several ways significantly more restrictive than OCaml's type system. In particular, in an inductive type definition, Rocq requires every constructor to satisfy a *positivity condition*. Furthermore, in an inductive function definition, Rocq requires the actual argument of every recursive call to be *structurally smaller* than the formal parameter of the function.

Because the formal definition of the positivity condition is quite difficult to read and understand, we provide a simplified paraphrase of it (Sections 6.1 and 6.2), followed with three simple examples of inductive type definitions that are rejected by Rocq (Section 6.3). Then, we point out why the OCaml definition of cadeques that we have presented in the previous section is similar to each of these three examples (Section 6.4). Fortunately, by parameterizing our type definitions with extra "level" and "size" indices, it is possible to work around all three problems: we sketch this technique, which we later exploit in a revisited implementation of non-catenable deques (Section 7) and in an implementation of catenable deques (Section 8).

We end this section with a discussion of a situation where the definition of a model function is rejected by Rocq because it is not structurally recursive— that is, this inductive function definition involves recursive calls whose arguments are not structurally smaller than the function's formal parameter. We are able to work around this problem by defining a model function at a more general type (Section 6.5). This idea is also later exploited in Sections 7 and 8.

6.1. **Uniform and non-uniform parameters.** Rocq splits the parameters of an inductive type in two groups, namely the uniform parameters and the non-uniform parameters. The distinguishing feature of a uniform parameter is that it must remain unchanged in every recursive occurrence of the type that is being defined.

For example, in the usual inductive definition of lists, which fits in one line under the form `Inductive list A := nil : list A | cons : A -> list A -> list A`, the parameter `A` is uniform.

On the contrary, in the definition of `plist A`, which appears in Fig. 27, the parameter `A` must be non-uniform. Indeed, in the type of the constructor `PCons`, the parameter `A` is instantiated with `A * A`. Thus, a parameter that is subject to recursive slowdown must be non-uniform.

A type with a non-uniform parameter, such as `plist`, is known in the literature as a *non-regular data type*, or a *nested data type*. The type `plist` is known as *Nest* by Bird and Meertens [BM98] and as *PList*, for *power list*, by Abel, Matthes, and Uustalu [AMU05].

The notion of uniform parameter plays a role in the statement of the positivity condition, which we now recall.

6.2. **The positivity condition.** A formal definition of the positivity condition appears in Rocq's reference manual. It is difficult to read. We offer the following simplistic paraphrase:

(PC) In the definition of an inductive type T,
if a constructor carries an argument of type U,
then the type T may occur within the type U at the following positions,
and *only* at these positions:
- at the root,
- in the right-hand side of a universal quantifier,
- inside a *uniform* argument of a *pre-existing* inductive type.

As a consequence, the positivity condition implies the following restriction:

(PR) In the definition of an inductive type T,
if a constructor carries an argument of type U,
the type T *cannot* occur within the type U
(PR1) inside a non-uniform argument of a pre-existing inductive type,
(PR2) inside an argument to itself.

```
Inductive tree (A : Type) : Type :=           (* accepted *)
  | Node : A -> list (tree A) -> tree A.

Inductive plist (A : Type) : Type :=          (* nested; accepted *)
  | PNil : plist A
  | PCons : A -> plist (A * A) -> plist A.

Fail Inductive ptree (A : Type) : Type :=     (* rejected *)
  | PNode : A -> plist (ptree A) -> ptree A.

Fail Inductive bush (A : Type) : Type :=      (* truly nested; rejected *)
  | Leaf : bush A
  | Cons : A -> bush (bush A) -> bush A.

Fail Inductive nztree (A : Type) : Type :=    (* rejected *)
  | NZLeaf : nztree A
  | NZNode : A -> forall ts: list (nztree A), length ts > 0 -> nztree A.
```

FIGURE 27. Three examples of inductive types that Rocq rejects

Although these statements are admittedly very informal, they help explain why the examples that we are about to present are rejected by Rocq. A better explanation of the strict positivity condition and of its interaction with nested inductive types is provided by Lamiaux et al. [LFST25].

6.3. **Examples of invalid inductive type definitions.** Fig. 27 presents two examples of inductive type definitions that Rocq accepts, followed with three examples that it rejects.

The type `tree` is a fairly common type of trees where each node carries an element of type `A` and a list of subtrees. Its definition is accepted. The fact that `tree A` occurs as an argument of the pre-existing inductive type `list` is not a problem because `A` is a uniform parameter of `list A`.

The type `plist`, already discussed earlier (Section 6.1), is the type of power lists. Its definition is accepted.

The definition of `ptree` is a copy of the definition of `tree`, where `list` has been replaced with `plist`. This definition is rejected because it violates (PR1). Indeed, `A` is a non-uniform parameter of `plist A`.

The definition of the type `bush` can be viewed as a copy of the definition of `plist` where the product _ * _ has been replaced with `bush` itself. This definition is rejected because it violates (PR2). As far as we know, the type *Bush* has first been defined by Bird and Meertens [BM98]. It is known by [AMU05] as a *truly nested data type* and by [BMPT17] as a *self-nested data type*.

The definition of the type `nztree` can be viewed as a variant of the definition of `tree`, where every node is required to carry a nonempty list of subtrees. (A new constructor for leaves is also introduced, so as to ensure that the type `nztree A` is nonempty.) This nonemptiness requirement is expressed by letting the constructor `NZNode` carry an argument of type `length ts > 0`. This type is convertible with `1 <= length ts`. This is sugar for `1 <= @length (nztree A) ts`, where the modifier `@` makes all arguments explicit. In this

```
Fail Inductive stored A : Type :=
  | Small :          list A -> stored A
  | Big   : chain (stored A) -> stored A

with chain A : Type :=
  | Single:        stored A -> chain A.
```

```
(* [istored A l] is meant to be isomorphic to [stored^l A]. *)
Inductive istored A : nat -> Type :=
  | Ground     :                  A -> istored A 0
  | Small  {l} : list (istored A l) -> istored A (S l)
  | Big    {l} :     ichain A (S l) -> istored A (S l)

(*  [ichain A l] is meant to be isomorphic to [chain (stored^l A)]. *)
with ichain A : nat -> Type :=
  | Single {l} :    istored A (S l) -> ichain A l.
```

FIGURE 28. Encoding a truly nested data type via indexing by levels

form, it is apparent that `nztree A` occurs inside the second argument of the relation `<=`. Furthermore, the relation `<=` is inductively defined, and a look at its definition shows that its second parameter is a non-uniform parameter. Therefore, the definition of `nztree` violates (PR1). It is rejected.

6.4. **Why our definitions exhibit similar features, and how to fix them.** Our OCaml definition of cadeques (Section 5), which involves several mutually recursive data types, exhibits similarities with each of the three examples of invalid inductive definitions that we have just presented. We point out these similarities and explain how we work around the problematic limitations of Rocq.

6.4.1. *A `bush`-like example.* Let us begin by considering the data types shown in Fig. 28. The upper part of the figure presents an attempt to define two mutually inductive types, `stored` and `chain`. This definition is rejected by Rocq because, like `bush` in Fig. 27, these data types are truly nested: in the type of the constructor `Big`, the parameter `A` is instantiated with `stored A`.

The data types `stored` and `chain` in Fig. 28 can be viewed as simplified versions of the types that we wish to define. Indeed, they are similar to the OCaml data types that we have presented in Fig. 24. The constructors `Big` and `Small` are roughly the same in both figures. Among the arguments of the constructor `Big`, we have kept just the child chain. In the argument of the constructor `Small`, we have represented the buffer as a list, for simplicity. In the definition of `chain`, we have performed a massive simplification, retaining just the fact that `chain A` depends on `stored A`.

The lower part of Fig. 28 shows how we work around the problem. We exploit the fact that the desired type definitions have a simple pattern of recursive slowdown: when one goes down by one level, the type parameter `A` is instantiated with `stored A`. The key idea, then, is to index every data type with a natural number `l`, which we refer to as a *level*. Intuitively, this natural number indicates how many times `stored`, viewed as a function of types to types, must be iterated. Thus, the types `istored A l` and `ichain A l` are intended to be isomorphic to `stored^l A` and `chain (stored^l A)`, where `stored` and `chain` are the types that we wished to define in the first place, and where we informally write _^_ for the usual power function. A level can also be visually understood as a depth: in this data structure, at level 0, one finds elements of type `A`; at level 1, elements of type `stored A`; at level 2, elements of type `stored (stored A)`; and so on.

The lower part of Fig. 28 is obtained from the upper part of this figure via a systematic transformation. In this transformation, the type `stored` plays a distinguished role because it is the pattern of recursive slowdown; in other words, it is the type function that is iterated. Here is a precise informal description of this transformation:

- In its header line, every data type is renamed and becomes parameterized with a level. For example, `chain A` is renamed to `ichain A`, and its result type is changed from `Type` to `nat -> Type`.
- In the type of every data constructor, the following transformations are successively applied:
  - A universal quantification `{l}` is introduced;
  - the variable `A` is instantiated with `stored^l A`;
  - every occurrence of `stored (stored^l A)` is replaced with `stored^(S l) A`;
  - every occurrence of `stored^l A` is replaced with `istored A l`;
  - every occurrence of `chain (stored^l A)` is replaced with `ichain A l`.
- In the distinguished data type `istored`, the constructor `Ground : A -> istored A 0` is introduced.

This transformation eliminates nesting: `istored` and `ichain` are ordinary (non-nested) inductive types. The type parameter `A` is a uniform parameter of `istored` and `ichain`. Indeed, it does not vary: instead, only the level varies. The definition of `istored` and `ichain` is accepted by Rocq.

Our Rocq implementation of Kaplan and Tarjan's non-catenable deques (Section 7) relies on this transformation, which we apply manually. This is visible in Fig. 35, where all types are indexed with levels, and where the definition of the type `stored` includes a constructor `Ground`.

6.4.2. *A `ptree`-like example.* In Fig. 28, `list` appears in the type of the constructor `Small`, at lines 2 and 11. Let us now imagine that `list` is replaced with `deque`, the type of non-catenable deques (Section 3). Then, the lower part of the figure breaks. Because `A` is a non-uniform parameter of `deque A`, Rocq does not allow `istored A l` to appear as an argument of `deque` on line 11.

In this form, this example is analogous to `ptree` in Fig. 27. Furthermore, the types that we wish to define exhibit a similar feature. Indeed, the constructor `Small` in Fig. 24 does carry a deque as an argument.[16]

Fortunately, the technique of indexing with levels, which has been described above (Section 6.4.1), allows working around this problem. The type `deque A`, where the parameter `A` is non-uniform, is transformed into `ideque A l`, where the parameter `A` is uniform. In the type of the constructor `Small`, instead of `deque (istored A l)`, one uses `ideque (istored A l) 0`.

[16] The abstract type `('a, _ plus3) Buffer.t`, once unfolded, is synonymous with `'a deque`.

6.4.3. *An `nztree`-like example.* In the type of the constructor `Small`, we need not only to indicate that this constructor carries a buffer, but also to impose a size constraint: this buffer must have size at least 3. This is visible in the type of the constructor `Small` in Fig. 24, where the size of the buffer is `_ plus3`.

Therefore, in Rocq, one is tempted to decorate the constructor `Small` (Fig. 28, line 11) with a constraint of the form `size d >= 3`, where the function `size` computes the size of a buffer. (Here, whether the buffer is a list or a deque is irrelevant.) However, this addition makes the lower part of Fig. 28 similar to `nztree` in Fig. 27, and causes it to be rejected by Rocq.

Fortunately, there is a simple workaround: the type of buffers should be indexed with a size. Then, one can impose a size constraint without referring to a `size` function. This eliminates the problem.[17] Looking ahead, this solution is visible in the type of the constructor `Small` in Fig. 35. There, the size index is instantiated with `3 + q`, forcing the size of the buffer to be at least 3.

6.5. **Structural recursion and model functions.** Each of the data structures considered in this paper is intended to represent a sequence of elements. To make this intent explicit, for each such data structure, we wish to define a *model function*, which maps a data structure to the sequence of elements that this data structure represents. Since the shape of a data structure is usually described in Rocq by an inductive type definition, its model function is usually defined via an inductive function definition. For this definition to be accepted by Rocq, it must be structurally recursive: that is, the actual argument of every recursive call must be a subterm of the formal parameter of the function.

Unfortunately, when a data structure is described by a nested inductive type, its model function typically involves nested recursive calls, which Rocq does not accept. Fig. 29 offers a simple example of this situation. There, the inductive type `buffer1` describes a "buffer" of exactly one element. The model function `buffer1_seq` turns a buffer into a sequence of elements: the buffer `B1 x` is turned into the singleton sequence `[x]`. Then, the inductive type `tree1` is defined. It is identical to the type `tree` in Fig. 27, except that its definition uses `buffer1` where the definition of `tree` uses `list`. At lines 14 and 15 in Fig. 29, an attempt is made to define a model function for the type `tree1`.[18] There, `ts` has type `buffer1 (tree1 A)`: it is a buffer of trees. Thus, `buffer1_seq ts` is a list of trees. The (list) `map` function lets us apply `tree1_seq` to each tree in this list. This yields a list of lists, which `concat` flattens to a list. Unfortunately, this definition is rejected by Rocq. The manner in which `tree1_seq` is used within its own definition is not acceptable. Indeed, via `map`, `tree1_seq` is applied to every tree in the list `buffer1_seq ts`. Even though a human reader can see that these trees are in fact subterms of `ts`, they are not "structurally smaller" than `ts` in the strict sense that Rocq requires.

17 In fact, in our OCaml implementation, the type `('a, 'n) Buffer.t` *was* indexed with a lower bound `'n` on the size of the buffer. There, this was the only viable approach, because OCaml does not have dependent types. In Rocq, indexing with sizes and relying on a `size` function are two approaches that seem viable at first. However, indexing turns out to be the only viable approach.

18 There is an alternative approach to defining the function `tree1_seq`. One first equips the type `buffer1` with a function `buffer1_map`. Then, one defines `tree1_seq (Node1 x ts)` as `concat (buffer1_seq (buffer1_map tree1_seq ts))`. Though it might be possible, we have not been able to scale this approach to the complex data structures considered in this paper.

```
(* A data type of buffers of exactly one element. *)
Inductive buffer1 (A : Type) : Type :=
  | B1 : A -> buffer1 A.

(* This turns a buffer into a mathematical sequence. *)
Equations buffer1_seq {A} : buffer1 A -> list A :=
buffer1_seq (B1 x) := [x].

(* This is analogous to tree in Fig. 27. *)
Inductive tree1 (A : Type) : Type :=
  | Node1 : A -> buffer1 (tree1 A) -> tree1 A.
```

```
(* This model function is rejected. *)
Fail Equations tree1_seq : forall {A}, tree1 A -> list A :=
tree1_seq (Node1 x ts) := [x] ++ concat (map tree1_seq (buffer1_seq ts)).
```

```
(* So far, we have been trying to define seq functions, whose type is: *)
Definition seq_fun (D : Type -> Type) : Type :=
  forall A,  D A -> list A.

(* We must in fact define concat_map_seq (cmseq) functions, whose type is: *)
Definition cmseq_fun (D : Type -> Type) : Type :=
  forall T,
  (forall A,     T A  -> list A) ->
  (forall A,  D (T A) -> list A).

(* A cmseq function can be specialized to obtain a seq function. *)
Definition cmseq2seq {D} (cmseq : cmseq_fun D) : seq_fun D :=
  cmseq (fun A => A) (fun _ x => [x]).

(* A cmseq function for buffer1. *)
Equations buffer1_cmseq : cmseq_fun buffer1 :=
buffer1_cmseq T f A (B1 x) := f _ x.

(* A cmseq function for tree1. *)
Equations tree1_cmseq : cmseq_fun tree1 :=
tree1_cmseq T f A (Node1 x ts) :=
  f _ x ++ buffer1_cmseq _ (tree1_cmseq _ f) _ ts.

(* A seq function for tree1, obtained as a special case. *)
Definition tree1_seq : seq_fun tree1 := cmseq2seq tree1_cmseq.

(* tree1_cmseq is correct. *)
Lemma correct_tree1_cmseq T (f : forall A, T A -> list A) A (t : tree1 (T A)) :
  tree1_cmseq T f A t = concat (map (f A) (tree1_seq (T A) t)).
```

FIGURE 29. An example of a model function that Rocq rejects, and a workaround

The reader might wonder whether, instead of structural recursion, our model functions could be defined by well-founded recursion. One approach would be to rely on the usual ordering of the natural numbers, which is well-founded. We would then need to equip each of our data types with a *measure function*, in such a way that the measure of a component is strictly smaller than the measure of the whole. Unfortunately, we would then run into another instance of the same problem: defining measure functions requires structural recursion, and Rocq is likely to reject our measure functions for the same reason that it rejects our model functions. Another approach might be to define a well-founded ordering directly on our data structures, and to define our model functions by recursion over an accessibility witness [Ler24]. However, because our data types are non-uniform, we would be unable to use a homogeneous ordering of type `D A -> D A -> Prop`; instead we might need to use a heterogeneous ordering of type `D A -> D B -> Prop`. We have not explored this avenue.

To work around this obstacle, we define model functions of a more general type, which removes the need for explicit uses of list `concat` and list `map`. So far, for a data structure `D`, such as `buffer1` or `tree1`, we have defined model functions of type `forall A, D A -> list A`. That is, out of a data structure that contains elements, a (plain) model function extracts a sequence of elements. The idea is to generalize this as follows (lines 22–25). For an arbitrary notion of "troop", if out of a troop of elements one can extract a sequence of elements, then out of a data structure that contains troops of elements, a *generalized model function* extracts a sequence of elements.

A generalized model function can be specialized to obtain a plain model function: to do so, one lets a "troop" be a single element, so the "troop" type constructor is `fun A => A` and the sequence of elements of the troop `x` is the singleton sequence `[x]` (line 29).

Defining generalized model functions for the types `buffer1` and `tree1` is straightforward. In the definition of `tree1_cmseq` (line 38), the manner in which the recursive calls are nested reflects the manner in which the types `buffer1` and `tree1` are nested in the definition of `tree1` (line 11). There is no longer a need to use `concat` and `map`, as the effect of these functions has been built into the notion of a generalized model function. This definition is recognized by Rocq as structurally recursive: it is accepted.[19]

Out of the generalized model function `tree1_cmseq`, one recovers the plain model function `tree1_seq` that we wished to define in the first place (line 41). Furthermore, one can prove that `tree1_cmseq` is *correct* in the sense that it is equal to a composition of `concat`, `map`, and `tree1_seq` (lines 44 and 45). The proof requires an induction principle for the type `tree1`.

In this paper, we define generalized model functions for non-catenable deques and for some of the data types involved in the definition of catenable deques (Section 8.2).

7. Non-catenable deques: revisited Rocq implementation

As explained in Section 6.4, the Rocq implementation of non-catenable deques that we have presented in Section 3.3 does not form a suitable building block for an implementation of catenable deques. Therefore, it must be revisited: every inductive type must be indexed by a level l and/or by a size n. For example, the type `chain A C` of Section 3.3, where `A` is the type of the elements and `C` is a color, becomes `chain A l n C` in this section. A chain of this type stores n elements of type A^{2^l}, that is, $n \cdot 2^l$ elements of type A.

[19] In order to see that this definition is acceptable, one must destruct the buffer `ts` and inline away `buffer1_cmseq`. Then, it becomes clear that `tree1_cmseq` is applied to a subterm of `Node1 x ts`.

```
Definition level := nat.
Definition size := nat.

Inductive prodN (A : Type) : level -> Type :=
  | prodZ     : A -> prodN A 0
  | prodS {l} : prodN A l -> prodN A l -> prodN A (S l).

Inductive buffer (A : Type) (l : level) : size -> color -> Type :=
  | B0         : buffer A l 0 red
  | B1 {y r}   : prodN A l -> buffer A l 1 (Mix NoGreen y r)
  | B2 {g y r} : prodN A l -> prodN A l -> buffer A l 2 (Mix g y r)
  | B3 {g y r} : prodN A l -> prodN A l -> prodN A l ->
                 buffer A l 3 (Mix g y r)
  | B4 {y r}   : prodN A l -> prodN A l -> prodN A l ->
                 prodN A l -> buffer A l 4 (Mix NoGreen y r)
  | B5         : prodN A l -> prodN A l -> prodN A l ->
                 prodN A l -> prodN A l -> buffer A l 5 red.

Inductive packet (A : Type) (l : level) : level -> size -> size -> color -> Type :=
  | Hole {n}                      :
      packet A l l n n uncolored
  | Packet {hl pn pktn sn hn C y} :
      buffer A l pn C ->
      packet A (S l) hl pktn hn (Mix NoGreen y NoRed) ->
      buffer A l sn C ->
      packet A l hl (pn + 2 * pktn + sn) hn C.

Inductive chain (A : Type) (l : level) : size -> color -> Type :=
  | Ending {n C}          :
      buffer A l n C ->
      chain A l n green
  | Chain {hl n hn C1 C2} :
      regularity C1 C2 ->
      packet A l hl n hn C1 ->
      chain A hl hn C2 ->
      chain A l n C1.

Inductive deque (A : Type) (n : size) : Type :=
  | T {g y} : chain A 0 n (Mix g y NoRed) -> deque A n.
```

FIGURE 30. Deques (Rocq): level- and size-indexed types for deques

The new (indexed) type definitions appear in Fig. 30. They can be compared with the earlier (non-indexed) type definitions in Fig. 13.

These indexed definitions are obtained by manually applying the transformation that was sketched in Section 6.4.1. Here, the "pattern of recursive slowdown" is the product type `_ * _`. Indeed, when one goes down by one level in the data structure, the parameter `A` is instantiated with `A * A`. Thus, the index l indicates how many times the product type constructor is iterated.

7.1. **Iterated products.** Rocq's product type, written `A * B` or `prod A B`, is an inductive type with one constructor, `pair`. By specializing it to the homogeneous case, where both components have type `A`, and by applying the transformation to it, one obtains the inductive type `prodN` in Fig. 30. This is an iterated product type: a value of type `prodN A l` consists of 2^l elements of type `A`. The constructor `prodZ` is introduced by the transformation: it corresponds to the `Ground` constructor in Section 6.4.1. The constructor `prodS` is an indexed version of the pre-existing constructor `pair`.

7.2. **Buffers.** The type `buffer` is transformed in a straightforward way. It becomes indexed by a level and a size (Fig. 30). In comparison with the non-indexed version of this type in Fig. 13, an atomic element of type `A` is replaced with a composite element of type `prodN A l`. Furthermore, the size index n reflects the number of composite elements in the buffer. Thus, a buffer of type `buffer A l n c` stores n elements of type A^{2^l}.

7.3. **Packets.** The type `packet` is indexed with levels in a slightly ad hoc manner. In the original type `packet A B C` (Fig. 13), A was the type of the elements at the top level of the packet and B was the type of the elements at the bottom level of the packet.[20] By definition of the type `packet`, the parameter B was implicitly constrained to always be of the form A^{2^d}, where d is the depth of the packet, that is, the difference between the bottom level and the top level. Here, we choose to abandon the parameter B and instead keep track of two levels, namely the top level and the bottom level of the packet. In the type `packet A l hl _ _ C`, the index `l` is the top level, while `hl` is the bottom level, or *hole level*. The level `hl` is also the top level of the next packet in the chain; this is visible in the type of the constructor `Chain` (Fig. 30).

Furthermore, we index the type `packet` with two sizes. In `packet A l hl n hn C`, the index `n` represents the total size of this packet and of the subchain that follows it, under the assumption that the size of the subchain is `hn`. (As noted earlier, by convention, the "size" of a data structure whose top level is `l` is the number of composite elements of type `prodN A l` that this data structure contains.) The formula `pn + 2 * pktn + sn` at line 26 in Fig. 30 adds up the size `pn` of the prefix buffer, the size `pktn` of the subpacket and of the subchain that follows it, and the size `sn` of the suffix buffer. The multiplication by two accounts for the fact that the size `pktn` is expressed in a different unit of measure: whereas this packet has top level `l`, its subpacket has top level `S l`. Because one element of type `prodN A (S l)` corresponds to two elements of type `prodN A l`, the size of the subpacket must be multiplied by two when it is viewed as a contribution to the size of the packet.

7.4. **Regularity.** The type `regularity` does not need to be indexed. It remains the same as in Fig. 13. Therefore, it is not shown again in Fig. 30.

7.5. **Chains.** The type `chain` becomes indexed by a level and by a size (Fig. 30). As announced earlier, a chain of type `chain A l n C` stores n elements of type A^{2^l}. In the type of the constructor `Chain`, at line 28 in Fig. 30, the hole level and hole size of the first packet are required to coincide with the top level and size of the subsequent chain.

[20] The parameter C is a color. The treatment of colors is unaffected by the introduction of indexing.

7.6. **Deques.** The type `deque` becomes indexed by a size (Fig. 30). For our purposes, there is no need to index it with a level. So, we define a deque as a chain whose top level is zero. If this chain has size n, then we declare that this deque has size n. This makes sense because the top level of the chain is zero: thus, it does not matter whether the size is measured as a number of atomic elements of type A or as a number of composite elements of type A^{2^0}.

7.7. **Code and proofs.** The introduction of level indices does not require any additional functions, lemmas, or proofs. However, our type definitions become more intricate: their size grows from 30 to 60 lines. For the sake of illustration, in our online repository, we provide an implementation of deques that is indexed with just levels.

In contrast, the introduction of size indices presents a greater challenge: new proof obligations appear, which are equalities between natural numbers. As a consequence, three new functions, two new lemmas, and a custom tactic are required. Our verified Rocq implementation of non-catenable deques, indexed with levels and sizes, spans about 670 lines, including 80 lines of type definitions and 30 lines of model function definitions.

8. Catenable deques: Rocq implementation

In the previous section (Section 7), we have presented a Rocq implementation of non-catenable deques. From here on, we refer to the type of non-catenable deques via its qualified name, `Deque.deque A n`. This type has two qualities that are important for our purposes in this section: its parameter `A` is a uniform parameter; and it is indexed with a size parameter `n`. These qualities allow us to use non-catenable deques (with size constraints) at the very heart of the inductive definition of catenable deques, while avoiding the obstacles discussed in Section 6. Furthermore, for this definition to be accepted by Rocq, its main data types must be indexed with levels, as explained in Section 6. Up to these differences, our Rocq implementation of catenable deques closely follows the OCaml implementation of Section 5.

8.1. **Types.** In a catenable deque, a *buffer* is a non-catenable deque. To describe buffers, we rely on the type `Deque.deque A n` of Section 7. As can be seen in Fig. 30, although the definition of this type relies on auxiliary data types that are indexed with levels and sizes, the type `Deque.deque A n` is indexed with just a size `n`.

The 4-color scheme is expressed in Rocq in a straightforward way (Fig. 31): each of the four hues forms an inductive type with two constructors; a color is a tuple of four hue bits.

As in Section 5.2.1, an *arity* is 0, 1, or 2, and a *kind* is “only”, “left”, or “right”. Here, kinds form an inductive type with three constructors, namely `only`, `left`, and `right` (Fig. 32). We define `arity` as a synonym for `nat` and define `empty`, `single`, and `pair` as short-hands for 0, 1, and 2 (Fig. 32).

The definitions of the types `node_coloring` and `node` appear in Fig. 33. These definitions are analogous to our earlier OCaml definitions (Fig. 23). The Rocq types `prefix'`, `suffix'`, and `node'` in Fig. 33 correspond to the OCaml types `prefix`, `suffix`, and `node` in Fig. 23. The reason why we use “primed” names is that we later define `prefix A~l`, `suffix A~l`, and `node A~l` as abbreviations for `prefix' (stored A~l)`, `suffix' (stored A~l)`, and `node' (stored A~l)`. This is convenient in our code, but is not visible in the paper.

The type `regularity` is easily translated from OCaml (Fig. 25) to Rocq (Fig. 34).

To translate the definition of the four inductive types `stored`, `body`, `packet`, and `chain` from OCaml (Fig. 24) to Rocq (Fig. 35), indexing with levels is used, as described in

```
(* Each of the four hues is an inductive type with two constructors. *)
Inductive green_hue  := SomeGreen  | NoGreen.
Inductive yellow_hue := SomeYellow | NoYellow.
Inductive orange_hue := SomeOrange | NoOrange.
Inductive red_hue    := SomeRed    | NoRed.

(* A color is a quadruple of four hue bits. *)
Inductive color    :=
  Mix : green_hue -> yellow_hue -> orange_hue -> red_hue -> color.

Notation green     := (Mix SomeGreen   NoYellow   NoOrange   NoRed).
Notation yellow    := (Mix   NoGreen SomeYellow   NoOrange   NoRed).
Notation orange    := (Mix   NoGreen   NoYellow SomeOrange   NoRed).
Notation red       := (Mix   NoGreen   NoYellow   NoOrange SomeRed).
Notation uncolored := (Mix   NoGreen   NoYellow   NoOrange   NoRed).
```

FIGURE 31. Cadeques (Rocq): types for colors

```
(* An arity is 0, 1, or 2. *)
Notation arity  := nat.
Notation empty  := 0.
Notation single := 1.
Notation pair   := 2.

(* A kind is only, left, or right. *)
Inductive kind : Type := only | left | right.

(* A level is a natural integer. *)
Definition level := nat.

(* A size is a natural integer. *)
Definition size := nat.
```

FIGURE 32. Cadeques (Rocq): arities, kinds, levels, and sizes

Section 6.4.1. Here, the pattern of recursive slowdown is `stored _`. Thus, a new constructor `Ground : A~-> stored A~0` appears in the definition of the type `stored`. The type `packet` is translated in the same ad hoc way as in Section 7.3, so it becomes indexed with two levels. The last definition is the definition of the type `cadeque` (Fig. 36); it is analogous to the type `cadeque` in Fig. 26.

Our type definitions include 24 types in total and span 190 lines of code.

8.2. **Model functions.** In order to be able to state the correctness of the operations on catenable deques, we must define model functions.

Following the approach outlined in Section 6.5, for the non-catenable deques of Section 7, we define the generalized model function `Deque.deque_cmseq` (not shown). For the type `node'` in Fig. 33, we also define a generalized model function, `node'_cmseq` (not shown). For

```
(* Convenient synonyms. *)
Definition prefix' := Deque.deque.
Definition suffix' := Deque.deque.

(* Constraints on buffer sizes, arity, and color. *)
Inductive node_coloring : size -> size -> arity -> color -> Type :=
  | EN {qp qs}   : node_coloring (0 + qp) (0 + qs) 0     green
  | GN {qp qs a} : node_coloring (3 + qp) (3 + qs) (S a) green
  | YN {qp qs a} : node_coloring (2 + qp) (2 + qs) (S a) yellow
  | ON {qp qs a} : node_coloring (1 + qp) (1 + qs) (S a) orange
  | RN {qp qs a} : node_coloring (0 + qp) (0 + qs) (S a) red.

(* Nodes. *)
Inductive node' (A : Type) : arity -> kind -> color -> Type :=
  | Only {qp qs a C}  :
      node_coloring qp qs (S a) C ->
      prefix' A (5 + qp) ->
      suffix' A (5 + qs) ->
      node' A (S a) only C
  | Only_end {q}      :
      prefix' A (S q) ->
      node' A 0 only green
  | Left {qp qs a C}  :
      node_coloring qp qs a C ->
      prefix' A (5 + qp) ->
      A * A ->
      node' A a left C
  | Right {qp qs a C} :
      node_coloring qp qs a C ->
      A * A ->
      suffix' A (5 + qs) ->
      node' A a right C.
```

Figure 33. Cadeques (Rocq): types for nodes

```
Inductive regularity : color -> color -> color -> Type :=
  | G {lC rC} : regularity green lC    rC
  | R         : regularity red   green green.
```

Figure 34. Cadeques (Rocq): regularity constraints

the four mutually inductive types `stored`, `body`, `packet`, and `chain`, four mutually recursive model functions are needed. For these types, although we could define generalized model functions, it is possible and sufficient to define plain model functions, shown in Fig. 37.

The definitions of the model functions for catenable deques amount to 110 lines in total. Rocq type-checks them in approximately 30 minutes. The reason why so much time is required is not clear to us; we have reported a performance issue.[21]

[21] https://github.com/mattam82/Coq-Equations/issues/620

```
Inductive stored (A : Type) : level -> Type :=
  | Ground                :
      A -> stored A 0
  | Big {l qp qs a lC rC} :
      prefix' (stored A l) (3 + qp) ->
      chain A (S l) a only lC rC ->
      suffix' (stored A l) (3 + qs) ->
      stored A (S l)
  | Small {l q}           :
      suffix' (stored A l) (3 + q) ->
      stored A (S l)

with body (A : Type) : level -> level -> kind -> kind -> Type :=
  | Hole {l k}                    :
      body A l l k k
  | Single_child {hl tl hk tk y o}:
      node' (stored A hl) 1 hk (Mix NoGreen y o NoRed) ->
      body A (S hl) tl only tk ->
      body A hl tl hk tk
  | Pair_yellow {hl tl hk tk C}   :
      node' (stored A hl) 2 hk yellow ->
      body A (S hl) tl left tk ->
      chain A (S hl) single right C C ->
      body A hl tl hk tk
  | Pair_orange {hl tl hk tk}     :
      node' (stored A hl) 2 hk orange ->
      chain A (S hl) single left green green ->
      body A (S hl) tl right tk ->
      body A hl tl hk tk

with packet (A : Type) : level -> level -> arity -> kind -> color -> Type :=
  | Packet {hl tl a hk tk g r} :
      body A hl tl hk tk ->
      node' (stored A tl) a tk (Mix g NoYellow NoOrange r) ->
      packet A hl (S tl) a hk (Mix g NoYellow NoOrange r)

with chain (A : Type) : level -> arity -> kind -> color -> color -> Type :=
  | Empty {l k lC rC}          :
      chain A l empty k lC rC
  | Single {hl tl a k C lC rC} :
      regularity C lC rC ->
      packet A hl tl a k C ->
      chain A tl a only lC rC ->
      chain A hl single k C C
  | Pair {l k lC rC}           :
      chain A l single left lC lC ->
      chain A l single right rC rC ->
      chain A l pair k lC rC.
```

Figure 35. Cadeques (Rocq): types for stored triples, bodies, packets, chains

```
Inductive cadeque : Type -> Type :=
  | T {A a} : chain A 0 a only green green -> cadeque A.
```

FIGURE 36. Cadeques (Rocq): the type of cadeques

```
Equations stored_seq A l (st : stored A l) : list A
by struct st :=
stored_seq A l (Ground a) :=
  [a];
stored_seq A l (Small s) :=
  Deque.deque_cmseq stored_seq s;
stored_seq A l (Big p child s) :=
  Deque.deque_cmseq stored_seq p ++
  chain_seq child ++
  Deque.deque_cmseq stored_seq s

with body_seq {A hl tl hk tk} (b : body A hl tl hk tk) : list A -> list A
by struct b :=
body_seq Hole accu :=
  accu;
body_seq (Single_child hd b) accu :=
  node'_cmseq stored_seq hd (body_seq b accu);
body_seq (Pair_yellow hd b cr) accu :=
  node'_cmseq stored_seq hd (body_seq b accu ++ chain_seq cr);
body_seq (Pair_orange hd cl b) accu :=
  node'_cmseq stored_seq hd (chain_seq cl ++ body_seq b accu)

with packet_seq {A hl tl ar k C} : packet A hl tl ar k C -> list A -> list A :=
packet_seq (Packet b tl) accu :=
  body_seq b (node'_cmseq stored_seq tl accu)

with chain_seq {A l ar k lC rC} (c : chain A l ar k lC rC) : list A
by struct c :=
chain_seq Empty := [];
chain_seq (Single _ pkt rest) := packet_seq pkt (chain_seq rest);
chain_seq (Pair lc rc) := chain_seq lc ++ chain_seq rc.

Equations cadeque_seq {A} : cadeque A -> list A :=
cadeque_seq (T c) := chain_seq c.
```

FIGURE 37. Cadeques (Rocq): model functions

8.3. **Code.** The operations on catenable deques are ported from OCaml to Rocq in a rather straightforward way. About 60 auxiliary functions are involved, on top of which the main five operations, *push*, *inject*, *pop*, *eject*, and *concat*, are easily defined.

The correctness of each auxiliary function or operation is expressed by decorating its definition with a postcondition whose statement uses suitable model functions. For example, the correctness of *push* is expressed by decorating the definition of `push x d` with the postcondition `cadeque_seq d' = [x] ++ cadeque_seq d`.

The auxiliary functions occupy approximately 1000 lines of code, while the main five operations span about 60 lines of code. All of these definitions are type-checked and verified by Rocq in a few minutes.

9. Executing our code

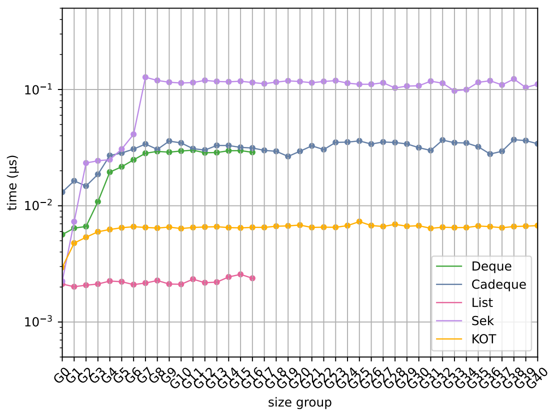
In this section, we discuss some practical aspects of our OCaml and Rocq implementations of catenable deques. We explain how we have tested our OCaml implementation and provide an evaluation of its performance (Section 9.1). Our Rocq implementation can in principle be executed either inside Rocq or outside Rocq, via a translation to OCaml (Section 9.2). Both approaches suffer from serious limitations, which destroy the worst-case constant-time complexity bound of our code, and are likely to have a severe impact on performance. We discuss these problems, but do not offer a numerical performance evaluation.

9.1. **OCaml implementation.** Although our OCaml code is not verified, we have experimentally validated its functional correctness via heavy random testing. Using Monolith [Pot21], we have built millions of well-typed scenarios, that is, sequences of operations of bounded length. We have run each scenario using two distinct implementations of catenable deques, namely our candidate implementation and a simplistic reference implementation, and we have checked that all observable results are the same. No discrepancies have been found.

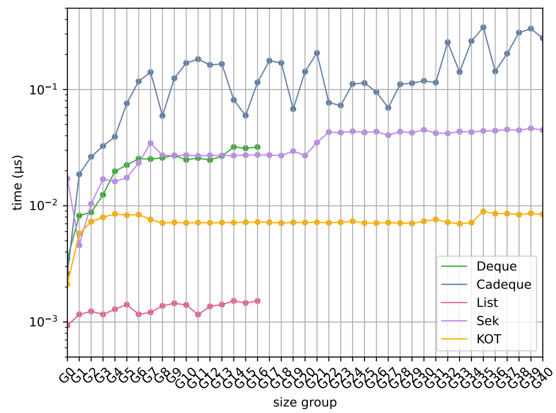
Furthermore, we have experimentally evaluated the performance of our OCaml code. For comparison, we have evaluated four persistent data structures that represent sequences:

(1) Kaplan and Tarjan's purely functional non-catenable deques (Section 3.2) ("Deque");
(2) Kaplan and Tarjan's purely functional catenable deques (Section 5) ("Cadeque");
(3) ordinary linked lists ("List");
(4) Charguéraud and Pottier's persistent chunked sequences [Sek] ("Sek");
(5) Kaplan, Okasaki, and Tarjan's simple persistent catenable deques [KOT00], which have been implemented and verified by Ponsonnet and Pottier [PP26] ("KOT").

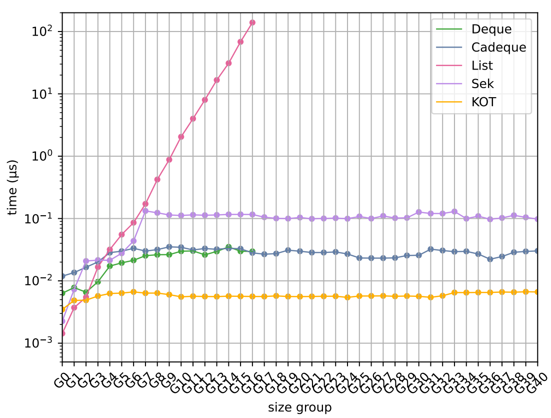
Our methodology is as follows. Because the data structures that we wish to evaluate are persistent, we do not restrict our attention to "single-threaded" scenarios, where each data structure is used at most once as an operand. We consider complex scenarios, where the operand or operands of each operation can be chosen in an arbitrary manner among the results of the previous operations. Furthermore, because we wish to exhibit the relationship between the cost of an operation and the logical length of its result, we need to build sequences whose logical lengths span a large interval and are roughly evenly distributed across this interval. We use the interval $[0, 2^{40})$. Therefore, we construct sequences whose logical lengths range up to roughly one trillion, that is, 10^{12}. We divide this interval into 41 disjoint subintervals, or *bins*, namely $[0, 1)$, $[1, 2)$, $[2, 4)$, and so on, up to $[2^{39}, 2^{40})$. Through constrained random search, we construct a scenario of 2050 operations ($2050 = 41 \times 50$) in such a way that each bin is populated with 50 inhabitants, that is, 50 data structures whose logical length falls within this bin. Each operation in this scenario is one of the five operations *push*, *pop*, *inject*, *eject*, and *concat*. (We use a smaller proportion of *concat* operations because they tend to produce sequences whose logical length is huge.) We measure how much time each operation in this scenario requires, and display the results, summarizing each bin as one data point. Because these data structures are persistent, an operation can



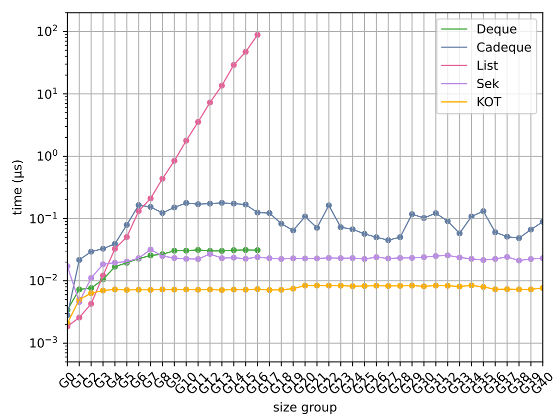
(A) *push*



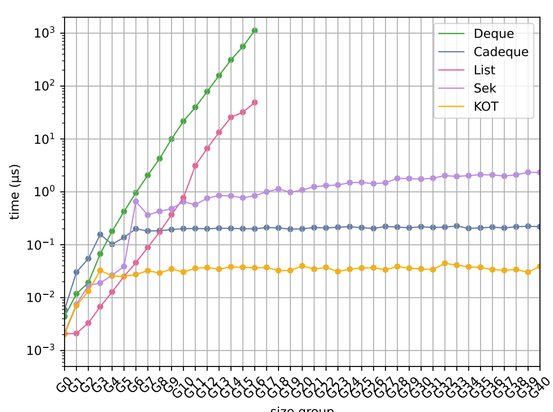
(B) *pop*



(C) *inject*



(D) *eject*



(E) *concat*, with arguments of roughly similar sizes

FIGURE 38. Experimental evaluation

be played and replayed in isolation as many times as one wishes. We take advantage of this feature to measure the cost of each operation independently and reliably.

If a persistent data structure has internal mutable state then this method is somewhat problematic: as an operation is replayed, its result remains the same, but its cost can vary. In our benchmark, this issue potentially affects Sek and KOT, both of which have internal mutable state. Sek is optimized to perform best under single-threaded scenarios, so we believe that, by replaying an operation several times, we are likely to observe the worst-case complexity. KOT is a self-adjusting data structure, which may update its internal state before answering a query; therefore we believe that the best-case complexity will likely be observed. We accept these shortcomings of our methodology because the main subjects of our benchmarks are List, Deque and Cadeque, which are purely functional. Our evaluation of Sek and KOT is provided for comparison but should not be considered fully reliable.

The benchmarks have been executed on a 2023 Apple MacBook Pro, equipped with an Apple M2 Max processor and 64 gigabytes of RAM. We used OCaml 5.4 and set the minor heap size to 512 million words, that is, 4 gigabytes. (This setting speeds up the benchmarks that allocate a lot of memory, such as the List benchmark, by a factor of up to 12.)

Because the more naive data structures, namely List and Deque, do not scale well to sequences of length 2^{40}, we stop measuring their performance at length 2^{16}.

Our results appear in Fig. 38. They correspond to commit `444358e` in our repository. In each graph, the horizontal axis shows the logical length of the result of an operation; a base-2 logarithmic scale is used. The vertical axis shows the time consumed by this operation, in microseconds; a base-10 logarithmic scale is used. Here is our reading of these graphs:

- As expected, each of the five data structures under test appears to perform a *push* operation in constant time. Lists are cheapest, at about 2 nanoseconds per operation. KOT comes second, at approximately 6 nanoseconds per operation. Deque and Cadeque come third, at about 36 nanoseconds per operation. Sek is most expensive, at about 120 nanoseconds per operation. One can see that Sek uses a different representation of short sequences up to size 64. At size 64 and above, Sek represents a sequence as a tree of *chunks*, which are arrays of size 128. When a *push* operation is repeatedly applied to the same sequence, each operation requires allocating a new chunk and copying data into it; this explains why Sek appears slowest in this benchmark.
- As expected, each of the five data structures appears to perform *pop* in constant time. In this benchmark, catenable deques appear more expensive than non-catenable deques, and their behavior seems somewhat more erratic. On catenable deques, a *pop* operation can cost up to 330 nanoseconds; *pop* appears to be significantly more expensive than *push*.
- As expected, lists perform *inject* and *eject* in linear time. The other three data structures perform these operations in constant time, with costs that seem roughly comparable to the costs of *push* and *pop*.
- We have measured the cost of *concat* for two arguments of arbitrary (independent) lengths. However, this information is difficult to visualize. Therefore, we show the diagonal only: the graph in Fig. 38e represents the cost of concatenating two sequences of roughly similar lengths. As expected, lists and non-catenable deques perform *concat* in linear time. Sek visibly does not have constant time complexity: this seems consistent with its documentation, which indicates that the complexity of concatenation is $O(\log n)$ in "common cases" and $O(\log^2 n)$ in the worst case. The remaining two data structures perform concatenation in constant time. Catenable deques behave consistently, at roughly 220 nanoseconds per operation.

In summary, although catenable deques can be up to 100 times slower than lists on *push* and *pop* operations, they appear to perform all five operations in constant time, as expected, at a speed of roughly one to ten million operations per second on this machine.

9.2. **Rocq implementation.** There are two main ways of executing a piece of code that is written in the programming language Rocq. One approach is to run it inside Rocq: several commands exist for this purpose. The other approach is to use Rocq's *extraction* mechanism, which translates Rocq code into OCaml code. This OCaml code can then be linked with other (verified or unverified) OCaml code and compiled into executable code. The first approach must be followed if one wishes to use our catenable deques as part of a verified algorithm (say, a decision procedure) and execute this algorithm as part of a proof that Rocq can check. The second approach can be used if one wishes to use our catenable deques as part of a (verified or unverified) program that does not need to run inside Rocq. We discuss both approaches in turn (Section 9.1, Section 9.2).

We must point out that at this time, regardless of which approach is chosen, our Rocq implementation is *not* efficient. Indeed, our data types involve levels and sizes, which Rocq is unable to ignore or erase, because they are natural numbers, whose sort is `Type`. Although we like to think of levels and sizes as "indices" that have no influence on the computation, we currently have no way of expressing this intent and of taking advantage of it. Because of this, our operations on catenable deques perform (intuitively useless, yet costly!) computations on levels and sizes, therefore do *not* have worst-case constant time complexity.

9.2.1. *Executing the verified code inside Rocq.* We have experimentally verified that our Rocq implementation is able to "compute inside Rocq". In other words, given a closed Rocq code fragment that uses our catenable deques and has a concrete type such as (say) `list nat`, Rocq can compute a normal form,[22] and this normal form is a canonical form for the type `list nat`: it is a list of natural numbers.

This result may seem trivial, but in fact it is not. We found that, in our initial attempt, reduction could be blocked by opaque terms.[23] Our analysis revealed that these opaque terms could appear as part of proofs of equality between indices, such as levels and sizes. For example, viewing a deque of size $2n+2$ as a deque of size $2(n+1)$ requires a cast, which itself requires a proof of the equality $2n+2 = 2(n+1)$. When Rocq attempts to compute the normal form of a cast, it attempts to first reduce this proof of equality to a normal form,[24] and can then reduce the cast itself. If for some reason this proof of equality contains an opaque term, then the whole process becomes blocked.

In our initial attempt, we constructed equality proofs by using off-the-shelf Rocq tactics, such as `lia`, a decision procedure for integer linear arithmetic. However, the proof terms produced by `lia` contain opaque terms. In fact, even the basic arithmetic lemmas found

22 This computation can be requested via the command **`Eval`** `vm_compute`.

23 A Rocq definition is either *transparent* or *opaque*. An opaque definition cannot be unfolded by Rocq's reduction engine. By default, all proofs are opaque. This is a good default because, in most uses of Rocq, there is no need to reduce proofs. Making proofs opaque saves time by forbidding reductions inside proofs, which would be pointless and could be costly. Here, though, the code that we wish to execute contain casts, which contain proofs of equality. These opaque proofs block execution.

24 This seems silly, as Rocq's meta-theory guarantees that every *closed* proof of equality can be reduced to `eq_refl`. However, as far as we understand, letting Rocq's type-checker assume that *every* proof of equality is convertible with `eq_refl` would break strong normalization and logical consistency.

```
Equations comp_eq {A} {eq_dec: EqDec A} {x y : A} (eq : x = y) : x = y :=
comp_eq eq with eq_dec x y => {
  | left e => e;
  | right ne => _ }.
```

FIGURE 39. Transforming a proof of an equality into a transparent proof of the same equality

in Rocq's standard library are declared opaque, because they are not intended to be used in computations. One could in principle define transparent copies of all of these lemmas, effectively duplicating part of the standard library, and either avoid the use of `lia` or define a copy of `lia` that uses our transparent copies of the basic lemmas, but that would be unbearably tedious. Clearly, we must use the standard library and `lia` without modification.

To solve this issue, we use a clever trick that was communicated to us by Guillaume Melquiond. The key idea is to define a function, named `comp_eq` (Fig. 39), which transforms an arbitrary proof of the equality `x = y`, which may contain opaque terms, into another proof of *the same equality*, which does not contain opaque terms, so that Rocq is able to reduce this new proof to a normal form. The implementation of `comp_eq` is simple but clever. If `x` and `y` have type `A`, then one requires the type `A` to be equipped with an equality test, `eq_dec`, whose type is `forall x y : A, {x = y} + {x <> y}`. One applies this equality test to `x` and `y`. Crucially, the term `eq_dec x y` *can* (usually) be reduced to a normal form without becoming blocked somewhere along the way, because an equality test is intended for this very purpose. If this normal form is `left e`, where `e` is a proof of the equality `x = y`, then we return `e`. Furthermore, the case where this normal form is `right ne`, where `ne` is a proof of the disequality `x <> y`, cannot arise! Indeed, in this case, by confronting the equality proof `eq` and the disequality proof `ne`, we are able to obtain a contradiction. Thus, the second branch in the definition of `comp_eq` is dead. Because the equality proof `eq` is used only in this branch, there is never a need to reduce it to a normal form. Therefore, even though it might contain opaque subterms, it cannot block reduction.

Thanks to `comp_eq`, we can continue to use `lia` and other off-the-shelf tactics to prove equalities between levels and sizes. We simply need to wrap the resulting proof with `comp_eq`. This amounts to ignoring the computational content of the equality proof produced by `lia` and replacing it with a direct equality test, which we know will always succeed.

Of course, this state of things is not entirely satisfactory. First, it requires the type of indices (in our case, `nat`) to be equipped with an equality test. Second, it forces us to actually execute this equality test (and pay its runtime cost), even though we have proved that it cannot fail. Finally, it requires the indices `x` and `y` to exist at runtime, even though we would like to think of them as computationally irrelevant.

9.2.2. *Extracting the verified code from Rocq to OCaml.* We have checked that our verified implementation can be translated to OCaml using Rocq's extraction mechanism. This attests that our Rocq formalization is indeed executable.

The resulting OCaml code has same the computational content as the Rocq code: all proofs are erased during extraction, but indices (levels and sizes) remain. Thus, the OCaml code that is obtained via extraction is not efficient and does not have worst-case time complexity $O(1)$. In practical applications, our hand-written OCaml code should be used.

10. Related work

10.1. **Families of persistent data structures.** Kaplan and Tarjan's purely functional, real-time catenable deques are part of a broader landscape of persistent data structures. For an overview of the previous work in this area, we refer the reader to their paper [KT99]. Let us merely indicate that the persistent data structures in the literature can be classified in several families, which rely on different implementation techniques.

One family of data structures, which includes Driscoll et al.'s persistent lists with catenation [DST94] and Buchsbaum and Tarjan's persistent deques [BT95], achieve persistence by relying on imperative updates and explicit management of version numbers. Due to the use of imperative updates, it seems safe to say that the data structures in this family are comparatively difficult to verify.

A second family, which includes Okasaki's simple queues and deques [Oka95], his catenable deques [Oka97], as well as a large variety of data structures described in his book [Oka99], involve suspensions—simple mutable data structures that offer a basic form of memoization. They can be easily and elegantly implemented in lazy functional programming languages, such as Haskell, where suspensions are implicit. These data structures are known as "purely functional" because memoization influences their time and space complexity but not their correctness. Therefore, in a proof of correctness, the presence of memoization, and the fact that memoization involves imperative updates, can be ignored.

A somewhat atypical member of this family is Kaplan, Okasaki, and Tarjan's simple catenable deques [KOT00]. Instead of relying on suspensions, whose mutable field can be written at most once, this data structure involves mutable fields that can be written several times. However, the authors ensure that every update is logically unobservable, because the overwritten value and the new value are two data structures whose logical models are equal. Again, these restricted updates influence the data structure's time and space complexity but not its correctness. The correctness and time complexity of this data structure have been verified by Ponsonnet and Pottier [PP26].

By ignoring the presence of imperative updates, the data structures in this second family can in principle be verified within a proof assistant based on type theory, such as Rocq. However, if the code is executed (either inside Rocq or via extraction to OCaml) without imperative updates then its time complexity will be incorrect.

A third family of data structures are "strictly functional": that is, they rely strictly on immutable heap-allocated memory blocks and pointers between these blocks. No imperative updates, whether explicit or implicit, are involved. Examples of this family include Kaplan and Tarjan's persistent catenable stacks [KT95] as well as the data structure that is considered in this paper, namely, Kaplan and Tarjan's persistent real-time catenable deques [KT99]. Many more examples can be found in the books by Appel [VFA], by Nipkow et al. [FAV], and by Nipkow [Nip25]. The data structures in this third family can be implemented and verified within a proof assistant based on type theory, such as Rocq. Furthermore, there is hope that, someday, these data structures can be efficiently executed inside Rocq or via extraction to OCaml. This requires irrelevant proofs and indices to be erased, which, today, does not yet seem possible (Section 9, Section 11).

10.2. **Kaplan and Tarjan's deques.** In this paper, we have described OCaml and Rocq implementations of Kaplan and Tarjan's real-time catenable deques [KT99], a strictly functional data structure. Kaplan and Tarjan provide only an English description of the data structure: therefore, the type definitions, the code, and the proofs of correctness are contributions of this paper.

A set of unpublished course notes by Mihaescu and Tarjan [MT03] sketches a variation of Kaplan and Tarjan's non-catenable and catenable deques. In comparison with Kaplan and Tarjan, Mihaescu and Tarjan appear to impose stricter size constraints on buffers. Their description is informal and brief. We have not studied it in depth. It would be interesting to implement it in OCaml, Rocq, or some other typed functional programming language. The techniques that we have used can in principle be applied to it.

An online repository by Thomas Refis,[25] dated 2013, contains a Rocq implementation of Kaplan and Tarjan's non-catenable deques. The data structure is represented as chain of packets, where the type of packets carries two parameters `A` and `B`, as in our code (Fig. 13). The data structure's invariant, which involves sizes and colors, is expressed a posteriori. There is a proof that the four operations preserve this invariant. There is no proof of functional correctness.

Schaub [Sch23] describes a Haskell implementation of Kaplan and Tarjan's non-catenable deques. This implementation is not verified. Schaub describes the data structure with a plain algebraic data type: it is not a generalized or nested algebraic data type. As a result, the content of a buffer must be described in a coarse way: it is a "binary tree" where every node is either a leaf (which carries an element) or a binary internal node. To distinguish between these cases, every node carries a tag, and runtime tests are needed. If the code contained a conceptual mistake, these tests could fail. We do not need such tags or runtime tests: by exploiting nested and generalized algebraic data types, we are able to represent the content of a buffer as nested pairs of type A^{2^l}.

Two online repositories by Alex Lang[26] and by Edward Kmett,[27] which are related to one another and are both dated 2015, appear to contain implementations of Kaplan and Tarjan's non-catenable and catenable deques. This code is not commented and appears to be untested.[28] It has not been released or verified.

10.3. **Connections with numeral systems.** The ties between numeral systems and data structures have been pointed out and exploited by many researchers, including Clancy and Knuth [CK77], Vuillemin [Vui78], Myers [Mye83], Hinze [Hin98], Okasaki [Oka99, Chapter 9]. Elmasry and Katajainen [EK22] survey the use of regular numeral systems in the design of data structures. In Kaplan and Tarjan's data structures [KT95; KT99], the redundant binary representation serves as a source of inspiration, but in the end, the connection between the numeral system and the data structure appears somewhat loose. In particular, a data structure that represents a sequence of n elements does not have the same shape as a representation of the number n.

10.4. **Nested data types.** Nested data types [BM98] play a central role in the description of the data structures considered in this paper and of many other data structures in the literature. The need for nested data types naturally arises out of an algorithmic design principle: one typically wishes to organize a data structure in several levels and to view

[25] https://github.com/trefis/deques/

[26] https://github.com/alang9/deque/

[27] https://github.com/ekmett/deque/

[28] Alex Lang, private communication, August 2024.

a group of elements at level l collectively as a single element at level $l + 1$. Thus, as one moves down from one level to the next, the type parameter that describes "elements" must be instantiated with "group of elements". As a suitable notion of "group", one might wish to use a pre-existing data type, such as "pair", or one of the data types involved in the data structure that is being defined, in which case a *truly nested* data type [AMU05] arises. The data type *Bush* [BM98] is an example of a truly nested data type; so are the data types of non-catenable deques and catenable deques considered in this paper.

As discussed in Section 6, defining and working with nested data types can be difficult. Difficulties appear not only in Rocq but also in other proof assistants, such as Agda. In some cases, the definition of a nested data type is rejected by the proof assistant; in other cases, it is accepted, but the definition of an inductive function over this data type is rejected. To work around these obstacles, we have indexed several types with levels and sizes (Section 6.4) and we have used generalized model functions instead of plain model functions (Section 6.5).

We have sketched a systematic way of transforming a data type into a level-indexed variant of this data type (Section 6.4.1). Although we could not find a paper where this transformation is documented, the idea of enriching nested data types with extra indices does appear in several forms in the literature. For example, Abel [Abe06] presents a system of "sized types" whose termination checker is type-based. This contrasts with Rocq's termination checker, which is based on an ad hoc syntactic criterion. Abel presents a "size"-indexed variant of the type *Bush* and shows that he is able to define inductive functions over this type. Besides, we remark that indexing *Bush* with a level seems tantamount to defining all powers of the type *Bush* simultaneously, an idea that was reportedly suggested to Matthes [Mat11] by Conor McBride.

In a related vein, Fu and Selinger [FS23] show that, although the type *Bush* can be defined in Agda (when the positivity check is turned off and when "large indices" are enabled), defining *map* and *fold* functions for this type is difficult. To resolve this difficulty, they define *map* and *fold* simultaneously for all powers of the type *Bush*. Thus, even though they do not index the type *Bush* with a level, they do in some places need to explicitly work with the type Bush^n, which can be viewed as a level-indexed variant of *Bush*.

Johann and Polonsky [JP20] criticize Fu and Selinger for working with an indexed variant of the type *Bush*, instead of directly with the type *Bush* itself. Still using Agda, they propose a different approach, *deep induction*, which is applicable to all algebraic data types, including nested types and truly nested types. They claim that this lets them present "the first-ever useful induction rule for bushes".

With a similar motivation, Montin, Ledein, and Dubois [MLD22] present a very general type, $LNDT$, of which many (but not all) nested data types, including *Bush*, are instances. They then proceed to define the functions *map* and *fold*, and to establish properties of these functions, in a generic way, for all instances of $LNDT$ at once. Unfortunately, the cyclic definition $\mathit{Bush} = LNDT\ \mathit{Bush}$ is rejected by Rocq, and is also rejected by Agda, unless its termination checker is turned off. As a possible work-around, they show that the type *Bush* can be indexed with a level (a special case of the systematic transformation that we have sketched), but they do not investigate this approach further.

Nested data types can also exist in systems where inductive data types are not a primitive feature, but must explicitly encoded in terms of lower-level concepts. For example, Blanchette et al. [BMPT17] encode nested data types in the proof assistant Isabelle/HOL, which is based on higher-order logic. Following an idea by Okasaki [Oka99, §10.1.1], which we have already discussed (Section 1.3.3), they perform a two-step construction. In the first step, an over-approximation of the desired nested data type is defined. In the second step, a natural-number-indexed well-formedness predicate is defined so as to select just the desired inhabitants. Then, the desired type is obtained as a subset type. Furthermore, Blanchette et al. are able to automatically construct induction and recursion principles for the type thus defined. They seem to indicate that their system does not support truly nested data types, such as *Bush*. It is not clear to us why this is so and whether this is an accidental or fundamental limitation.

11. Conclusion

11.1. **Summary.** In this paper, we have presented OCaml and Rocq implementations of Kaplan and Tarjan's non-catenable and catenable deques. Furthermore, we have verified the functional correctness of our Rocq implementations. Our proofs of functional correctness are mostly automated, therefore remarkably short: they occupy only 150 lines.

In this endeavor, the main challenge has been to translate Kaplan and Tarjan's textual descriptions of the data structures into type definitions. Beyond the initial hardship of understanding these complex data structures, we have faced the difficulty of dealing with the limited expressiveness of OCaml and Rocq. In OCaml, the absence of dependent types makes it difficult to reason about integer sizes at the type level; as a workaround, in one place, we have used a phantom type parameter. In Rocq, the positivity condition causes seemingly natural inductive type definitions to be rejected; as a workaround, we have indexed several types with integer levels or sizes.

In the end, thanks to OCaml's generalized algebraic data types and Rocq's indexed inductive types, we have been able to formulate type definitions that are precise, in the sense that they fully express the desired invariants. As we wrote the code of the main operations and of the many auxiliary functions, these type definitions offered an essential help: they enabled the early detection of several kinds of mistakes, including violations of an invariant and missing branches in case analyses.

11.2. **Lessons about Rocq.** From this work, we draw a few lessons about the Rocq proof assistant. On the positive side, the *Equations* plugin and the tactics *Hammer* and *AAC_tactics* are powerful and useful. We believe that, thanks to them, verifying purely functional data structures in Rocq can be a relatively pleasant experience.

On the negative side, Rocq as a programming language suffers from a certain lack of abstraction and modularity, which stems from the syntactic criteria that Rocq applies when determining whether an inductive type or function definition is acceptable. For example, in the course of this work, after defining what we thought was a nice abstract type `deque A` of non-catenable deques, we discovered that this type could not be used as planned inside the inductive definition of catenable deques. There were two problems with it. First, because `A` was a non-uniform parameter of `deque`, the inductive definition of catenable deques was not allowed to exhibit a cycle through this parameter. Second, even though we could define a function `size : deque A -> nat`, we were not allowed to use this function, within the inductive definition of catenable deques, to impose a constraint on the size of a deque. To work around these problems, we had to go back and index the type of non-catenable deques with integer levels and sizes. In fact, the type of catenable deques that we obtain in the end, `cadeque A`, suffers from a similar problem: it is not size-indexed. If someone wished to use a catenable deque, subject to a size constraint, in the definition of a new inductive type, then they may have to go back and define a size-indexed variant of the type `cadeque A`. For the sake of simplicity, we have not done this.

Another area where improvement seems needed is efficient execution of Rocq code, either within Rocq, or via extraction to OCaml. In either case, the integer indices that appear in our type definitions exist at runtime. They are computed at runtime and stored in the data structure. Furthermore, the code contains casts between indices, which are not erased, and have a significant runtime cost, as they require normalizing equality proofs. Notions of ghost data exists in several proof assistants and program verification environments, such as F*, Agda, Idris, and Why3. There is hope that Rocq, too, can be extended with such a notion [Win24]. We are very much looking forward to such a feature, which seems necessary for realistic (efficient) programming with indexed types.

11.3. **Future work.** In future work, it may be interesting to implement and verify the variation on catenable deques that is sketched in Mihaescu and Tarjan's course notes [MT03]. In a different direction, it could be a worthwhile exercise to use a program logic for OCaml, such as Osiris [SYMP25] or Zoo [All26], to prove that our OCaml code performs the same computations as our Rocq code. In essence, this would be a proof that the indices that exist in the Rocq code truly are "ghost" indices and are correctly erased in the hand-written OCaml code.

References

[Abe06] Andreas Abel. "Towards Generic Programming with Sized Types". In: *Mathematics of Program Construction (MPC)*. Vol. 4014. Lecture Notes in Computer Science. Springer, July 2006, pp. 10–28.

[All26] Clément Allain. "Zoo: A framework for the verification of concurrent OCaml 5 programs using separation logic". In: *Proceedings of the ACM on Programming Languages* 10.POPL (Jan. 2026).

[AMU05] Andreas Abel, Ralph Matthes, and Tarmo Uustalu. "Iteration and coiteration schemes for higher-order and nested datatypes". In: *Theoretical Computer Science* 333.1–2 (2005), pp. 3–66.

[BM98] Richard Bird and Lambert Meertens. "Nested Datatypes". In: *Mathematics of Program Construction (MPC)*. Vol. 1422. Lecture Notes in Computer Science. Springer, 1998, pp. 52–67.

[BMPT17] Jasmin Christian Blanchette, Fabian Meier, Andrei Popescu, and Dmitriy Traytel. "Foundational nonuniform (Co)datatypes for higher-order logic". In: *Logic in Computer Science (LICS)*. June 2017, pp. 1–12.

[BP11] Thomas Braibant and Damien Pous. "Tactics for Reasoning Modulo AC in Coq". In: *Certified Programs and Proofs (CPP)*. Vol. 7086. Lecture Notes in Computer Science. Springer, 2011, pp. 167–182.

[BT95] Adam L. Buchsbaum and Robert Endre Tarjan. "Confluently Persistent Deques via Data-Structural Bootstrapping". In: *Journal of Algorithms* 18.3 (1995), pp. 513–547.

[CK77] Michael J. Clancy and Donald E. Knuth. *A programming and problem-solving seminar*. Seminar report. Stanford University, Department of Computer Science, 1977.

[Cza20] Lukasz Czajka. "Practical Proof Search for Coq by Type Inhabitation". In: *International Joint Conference on Automated Reasoning*. Vol. 12167. Lecture Notes in Computer Science. Springer, July 2020, pp. 28–57.

[DLP10] Erik D. Demaine, Stefan Langerman, and Eric Price. "Confluently Persistent Tries for Efficient Version Control". In: *Algorithmica* 57.3 (2010), pp. 462–483.

[DST94] James R. Driscoll, Daniel Dominic Sleator, and Robert Endre Tarjan. "Fully Persistent Lists with Catenation". In: *Journal of the ACM* 41.5 (1994), pp. 943–959.

[EK22] Amr Elmasry and Jyrki Katajainen. "Regular numeral systems for data structures". In: *Acta Informatica* 59.2–3 (2022), pp. 245–281.

[Equations] The Rocq team. *Documentation of Equations*. Online tutorials. Jan. 2026.

[FAV] Tobias Nipkow, Jasmin Blanchette, Manuel Eberl, Alejandro Gómez-Londoño, Peter Lammich, Christian Sternagel, Simon Wimmer, and Bohua Zhan. *Functional Algorithms, Verified!* Apr. 2021.

[FS23] Peng Fu and Peter Selinger. "Towards an Induction Principle for Nested Data Types". In: *Workshop on Logic, Language, Information and Computation (WoLLIC)*. Vol. 13923. Lecture Notes in Computer Science. Springer, July 2023, pp. 244–255.

[Hin98] Ralf Hinze. *Numerical Representations as Higher-Order Nested Datatypes*. Technical Report IAI-TR-98-12. Institut für Informatik III, Universität Bonn, Dec. 1998.

[JP20] Patricia Johann and Andrew Polonsky. "Deep Induction: Induction Rules for (Truly) Nested Types". In: *Foundations of Software Science and Computation Structures (FOSSACS)*. Vol. 12077. Lecture Notes in Computer Science. Springer, Apr. 2020, pp. 339–358.

[KOT00] Haim Kaplan, Chris Okasaki, and Robert E. Tarjan. "Simple Confluently Persistent Catenable Lists". In: *SIAM Journal on Computing* 30.3 (2000), pp. 965–977.

[KT95] Haim Kaplan and Robert Endre Tarjan. "Persistent lists with catenation via recursive slow-down". In: *Symposium on the Theory of Computing*. May 1995, pp. 93–102.

[KT99] Haim Kaplan and Robert E. Tarjan. "Purely functional, real-time deques with catenation". In: *Journal of the ACM* 46.5 (1999), pp. 577–603.

[Ler24] Xavier Leroy. "Well-founded recursion done right". In: *Workshop on Coq for Programming Languages (CoqPL)*. Jan. 2024.

[LFST25] Thomas Lamiaux, Yannick Forster, Matthieu Sozeau, and Nicolas Tabareau. "Nested Inductive Types". Unpublished. 2025.

[LM99] Daan Leijen and Erik Meijer. "Domain-specific embedded compilers". In: *Conference on Domain-Specific Languages (DSL)*. Oct. 1999, pp. 109–122.

[Mat11] Ralph Matthes. "Map fusion for nested datatypes in intensional type theory". In: *Science of Computer Programming* 76.3 (2011), pp. 204–224.

[MLD22] Mathieu Montin, Amélie Ledein, and Catherine Dubois. "LibNDT: Towards a Formal Library on Spreadable Properties over Linked Nested Datatypes". In: *Workshop on Mathematically Structured Functional Programming (MSFP)*. Vol. 360. EPTCS. Apr. 2022, pp. 27–44.

[MMH22] Yaron Minsky, Anil Madhavapeddy, and Jason Hickey. *Real World OCaml: Functional programming for the masses, second edition*. Cambridge University Press, Oct. 2022.

[MT03] Radu Mihaescu and Robert E. Tarjan. "Notes on Catenable Deques in Pure Lisp". Unpublished course notes, COS 528. 2003.

[Mye83] Eugene W. Myers. "An Applicative Random-Access Stack". In: *Information Processing Letters* 17.5 (1983), pp. 241–248.

[Nip25] Tobias Nipkow. *Functional Data Structures and Algorithms: A Proof Assistant Approach*. ACM Books, Sept. 2025.

[OCaml] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The OCaml system, release 5.4*. Oct. 2025.

[Oka95] Chris Okasaki. "Simple and Efficient Purely Functional Queues and Deques". In: *Journal of Functional Programming* 5.4 (1995), pp. 583–592.

[Oka97] Chris Okasaki. "Catenable Double-Ended Queues". In: *International Conference on Functional Programming (ICFP)*. June 1997, pp. 66–74.

[Oka99] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1999.

[Pot21] François Pottier. "Strong Automated Testing of OCaml Libraries". In: *Journées Françaises des Langages Applicatifs (JFLA)*. Feb. 2021.

[PP26] Juliette Ponsonnet and François Pottier. "Verified Persistent Catenable Deques". In: *Journées Françaises des Langages Applicatifs (JFLA)*. Jan. 2026.

[Rey85] John C. Reynolds. "Three Approaches to Type Structure". In: *Theory and Practice of Software Development (TAPSOFT)*. Vol. 185. Lecture Notes in Computer Science. Springer, Mar. 1985, pp. 97–138.

[Rocq] The Rocq team. *The Rocq Prover*. 2026.

[Sch23] Fabian Schaub. *Purely Functional Real-Time Deques*. Seminar Report. Universität Innsbruck, 2023.

[Sek] Arthur Charguéraud and François Pottier. *Sek*. `https://gitlab.inria.fr/fpottier/sek`. 2020–2025.

[SM19] Matthieu Sozeau and Cyprien Mangin. "Equations reloaded: high-level dependently-typed functional programming and proving in Coq". In: *Proceedings of the ACM on Programming Languages*. Vol. 3. 2019, pp. 1–29.

[Soz07] Matthieu Sozeau. "Program-ing Finger Trees in Coq". In: *International Conference on Functional Programming (ICFP)*. Sept. 2007, pp. 13–24.

[SYMP25] Remy Seassau, Irene Yoon, Jean-Marie Madiot, and François Pottier. "Formal Semantics and Program Logics for a Fragment of OCaml". In: *Proceedings of the ACM on Programming Languages* 9.ICFP (Aug. 2025).

[VFA] Andrew W. Appel. "Verified Functional Algorithms". In: *Software Foundations*. Vol. 3. 2025.

[Vui78] Jean Vuillemin. "A Data Structure for Manipulating Priority Queues". In: *Communications of the ACM* 21.4 (1978), pp. 309–315.

[Win24] Théo Winterhalter. "Dependent Ghosts Have a Reflection for Free". In: *Proceedings of the ACM on Programming Languages* 8.ICFP (2024), pp. 630–658.

[XCC03] Hongwei Xi, Chiyan Chen, and Gang Chen. "Guarded Recursive Datatype Constructors". In: *Principles of Programming Languages (POPL)*. Jan. 2003, pp. 224–235.