

# *Fast thinnings via segment maps*

FRANÇOIS POTTIER

*Inria Paris, France*

(email: [francois.pottier@inria.fr](mailto:francois.pottier@inria.fr))

## **Abstract**

We present the *segment map*, a purely functional data structure that represents an increasing function of the natural numbers into the natural numbers, also known as a *thinning*. This data structure, a variant of Okasaki’s skew binary random access lists, offers efficient support for many operations, including lookup, reverse lookup, horizontal and vertical displacements, insertion at the front end, and composition. We present a verified implementation and an analysis of its time complexity. This data structure may be useful in proof assistants, compilers, and symbolic computation systems where variables are encoded as de Bruijn indices or co-de Bruijn indices.

## **1 Introduction**

This paper addresses a simple question: what is an efficient (purely functional) representation of *thinnings*, that is, increasing functions of the natural numbers to the natural numbers? Before offering an answer, let us take a step back and explain why this question is of interest.

### **1.1 Motivation**

Many software systems, such as compilers and proof assistants, manipulate abstract syntax. These systems must adopt a representation of variables that meets a number of needs, such as distinguishing free and binding occurrences of variables, comparing terms up to renamings of their bound variables, and renaming bound variables so as to avoid undesired name collisions.

Many representations of variables are known. They have often been studied primarily for their mathematical elegance, their safety, and their ease of use in verified software and in mechanized metatheory (Aydemir et al. 2005; Abel et al. 2019). However, in low-level implementations of realistic systems, where the main concern is efficiency, only a handful of representations appear to be commonly used. These are:

- (1) de Bruijn indices (de Bruijn 1972),
- (2) plain names (Peyton Jones and Marlow 2002; Cheney 2005),
- (3) the locally nameless representation, which combines de Bruijn indices and names (Gordon 1993; Aydemir et al. 2008; Charguéraud 2012),
- (4) techniques that exploit mutable data structures, such as circular doubly-linked lists and the union-find data structure, so as to achieve greater efficiency in specific scenarios (Appel and Jim 1997; Kennedy 2007).

In the realm of purely functional programming, much work has focused on finding safe and elegant ways of working with representations (1) to (3) above. This has given rise, among other ideas, to well-scoped de Bruijn representations (Bird and Paterson 1999), well-typed de Bruijn representations (Allais et al. 2021), well-scoped named representations (Maclaurin et al. 2023), as well as clever techniques that support the illusion of using a named representation while really using de Bruijn indices underneath (Bernardy and Pouillard 2013). Besides, much effort has been devoted to writing libraries and code generators that reduce

the burden and the danger of working with low-level representations of variables and binders. Some examples include LNgen (Aydemir and Weirich 2010), Unbound (Weirich et al. 2011), DBGen (Polonowski 2013), Needle & Knot (Keuchel et al. 2016), Tealeaves (Dunn et al. 2023), Rebound (Santo and Weirich 2025), and Sulfur (Bouverot-Dupuis et al. 2026).

Much of this work focuses on *offering a simple and safe user interface*, where “safety” means that a class of programming mistakes, such as forgetting to apply a renaming or constructing an incorrect renaming, is ruled out. However, relatively little published material addresses the subsequent problem of *also implementing this interface with a simple and cheap cost model*, where the operations that are exposed to the user’s view, such as constructing and deconstructing a term, have well-understood and low worst-case cost. This requires the operations that are removed from the user’s view, such as constructing, composing, and applying renamings, to be cheap.

The most straightforward implementations, which perform eager renamings, have well-understood cost, but this cost is high: the cost of constructing or deconstructing an abstraction (that is, a term that has a binder at its root) is typically linear in the size of this term. As a result, the time complexity of traversing or transforming a term is typically quadratic. Thus, it is generally recognized that, for greater efficiency, the syntax of terms should allow some form of *suspended* renamings (or substitutions). Furthermore, we note that the ability to *compose* two suspended renamings should somehow be exploited. Indeed, if composition is not used then the cost of applying renamings is delayed, but not avoided, and the time complexity of traversing or transforming a term remains quadratic. Calculi of explicit substitutions have been proposed and studied, both for their theoretical properties (Abadi et al. 1991; Kesner 2009) and for use in implementations (Nadathur and Qi 2003). Some of the libraries cited earlier, such as Rebound (Santo and Weirich 2025), use suspended substitutions.

Nevertheless, we believe that no previous work offers a representation of terms that enjoys both a simple and safe API and an implementation whose worst-case time complexity is guaranteed to be low. Indeed, many questions arise. To represent a renaming (or a substitution), what data structure should be used? In the syntax of terms, which nodes should carry a renaming? When is it cheaper to compose two renamings and when is it cheaper to apply them in sequence? Should some form of memoization be used, so as to avoid duplicating potentially expensive composition operations? How can one control the size and space usage of the suspended renamings? Should every subterm keep track of the set of its free variables, or of its maximum free de Bruijn index, so that the domains of renamings can be pruned? What is then the cost of pruning?

This paper does not address these questions, but is motivated by them. We focus on the very first question in the above list, namely, the problem of proposing an efficient (purely functional) representation of *thinnings*. A thinning (McBride 2018) is an increasing function of the natural numbers to the natural numbers. In a system where variables are represented as de Bruijn indices, thinnings can be used to represent a restricted class of renamings.

At this point, the reader might be tempted to debate our decision of restricting our attention to such a narrow problem. Why represent variables as de Bruijn indices? Why focus on representing renamings, which are functions of indices of indices, as opposed to substitutions, which are functions of indices to terms? Why focus on representing thinnings, which are increasing functions of indices to indices, as opposed to arbitrary renamings? We offer the following replies:

- (1) We believe that it may be worthwhile to work with a representation of terms where variables are de Bruijn indices and terms contain suspended thinnings.
- (2) Allowing just thinnings to be suspended, as opposed to arbitrary renamings or substitutions, is not necessarily restrictive. Many of the renamings whose need arises in practice are thinnings. The operation of eagerly applying an arbitrary substitution to a term can still be defined. It simply cannot be suspended.
- (3) A data structure that represents just thinnings, as opposed to arbitrary renamings or substitutions, can support some operations, such as reverse lookup and composition, more efficiently. When aiming to offer a representation of terms where every operation has guaranteed low cost, this could be crucial.

Through perhaps different paths, other authors have reached the conclusion that thinnings, and their performance, matter. Kovács (2023) uses thinnings, represented as bit vectors, in `smalltt`, a demonstration of “high-performance elaboration with dependent types”. Allais (2023) notes that thinnings are “omnipresent in the co-de Bruijn representation”, which makes their performance “a critical matter”.

## 1.2 Thinnings

We borrow the word “thinning” from McBride (2018), who attributes this notion to Altenkirch et al. (1995). In McBride’s view, a thinning is essentially a description of an increasing function of a finite interval  $[0, m)$  to a finite interval  $[0, n)$ . Let us write  $m \sqsubseteq n$  for the type of such a thinning. McBride gives an inductive definition of this type, of which we present a simplified form:

$$\begin{aligned}
 \text{Done} & : & 0 & \sqsubseteq 0 \\
 \text{Keep} & : & \forall mn. m \sqsubseteq n \rightarrow m + 1 & \sqsubseteq n + 1 \\
 \text{Drop} & : & \forall mn. m \sqsubseteq n \rightarrow m & \sqsubseteq n + 1
 \end{aligned}$$

McBride names these constructors `oz`, `_os`, and `_o'`, respectively. We borrow the names *Done*, *Keep*, and *Drop* from Allais (2023). The existence of a thinning  $\rho$  of type  $m \sqsubseteq n$  implies  $m \leq n$ . A thinning  $\rho$  of type  $m \sqsubseteq n$  is a description of an increasing function  $\llbracket \rho \rrbracket$  of  $[0, m)$  into  $[0, n)$ . This function is defined by the equations:

$$\begin{aligned}
 \llbracket \text{Keep } \rho \rrbracket(0) & = 0 \\
 \llbracket \text{Keep } \rho \rrbracket(i + 1) & = \llbracket \rho \rrbracket(i) + 1 \\
 \llbracket \text{Drop } \rho \rrbracket(i) & = \llbracket \rho \rrbracket(i) + 1
 \end{aligned}$$

This explains the names *Keep* and *Drop*. The constructor *Keep* means that the number 0 *is part* of the codomain of the function: it is kept. The constructor *Drop* means that the number 0 *is not part* of the codomain of the function: it is skipped, or dropped.

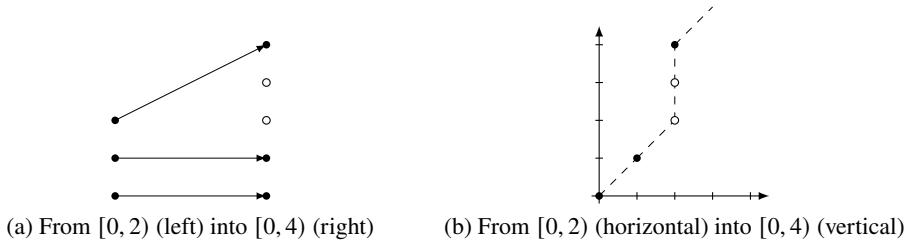


Fig. 1. Two graphical depictions of the thinning  $\text{Keep} (\text{Keep} (\text{Drop} (\text{Drop} (\text{Keep Done}))))$

To help visualize the meaning of the constructors *Keep* and *Drop*, Figure 1 presents two views of the thinning *Keep (Keep (Drop (Drop (Keep Done))))*. This thinning describes the increasing function of  $[0, 2)$  into  $[0, 4)$  that maps 0 to 0, 1 to 1, and 2 to 4. In Figure 1a, we display this function as a directed bipartite graph. Each edge connects a point in the domain (left) to a point in the codomain (right). In this view, the constructor *Keep* creates a black dot on each side and connects the two new dots with an edge. The constructor *Drop* creates a white unconnected dot on the right-hand side. The constructor *Done* is a terminator. In Figure 1b, we display the same function as a graph in the usual mathematical sense, that is, in the two-dimensional plane. The black dots at (0, 0), (1, 1), and (2, 4) are part of the function’s graph; the white dots at (2, 2) and (2, 3) are not. In this view, the constructors can be understood as “turtle graphics” instructions. *Keep* creates a black dot and moves towards the northeast by one unit. *Drop* creates a white dot and moves towards the north by one unit. *Done* is a terminator. In the remainder of this paper, we rely on the two-dimensional style of Figure 1b as a basis for intuition.

In summary, the inductive type of thinnings proposed by McBride (2018) can be understood as a list of diagonal segments of length 1 and vertical gaps of height 1. Although this inductive definition may be convenient for reasoning about thinnings in a dependently-typed programming language or proof assistant, such as Agda, Lean, Idris, or Rocq, it is clearly not as compact or as efficient as other representations. Several potential improvements on this representation come to mind. Let us briefly describe two alternative representations of thinnings, namely *bit vectors* and *segment lists*.

The basic idea behind bit vectors is simple. A thinning, as defined above, is a list of *Drop* and *Keep* constructors, terminated with *Done*. If *Drop* is encoded as 0 and *Keep* is encoded as 1 then a thinning can be encoded as a bit vector, that is, an array of bits. Allais (2023), working in Idris, defines thinnings both as an inductive type and as bit vectors and relates the two representations. Thus he is able to use the inductive view while reasoning and to use bit vectors at runtime, for greater efficiency. In a similar vein, Gonçalves and Kokke (2025) implement de Bruijn indices as machine words and thinnings as bit vectors in Haskell. Their library offers a safe, high-level API. The correctness of its unsafe, low-level implementation is ensured via testing.

Compared with the inductive representation, the bit vector representation is vastly more compact, and makes some operations faster by a large constant factor. However, in terms of *asymptotic* time complexity, it does not offer any improvement; in fact, it degrades the complexity of some operations. With both representations, the time complexity of lookup is  $O(n)$ , where  $n$  is the number of *Drop* and *Keep* constructors, or the length of the bit vector. With inductive thinnings, inserting a new diagonal segment (*Keep*) or a new vertical gap (*Drop*) at the front end of a thinning costs  $O(1)$ , whereas with bit vectors, the cost is  $O(n)$ , because the whole bit vector is copied.

A different way of improving the compactness of the representation of thinnings, while keeping an inductive representation, is to work with diagonal segments and vertical gaps whose length is at least 1 and can be greater than 1. Let the constructor *Keep* carry an integer argument *len*, which represents the length of a diagonal segment. Let the constructor *Drop* carry an integer argument *gap*, which represents the height of a vertical gap. Then, two consecutive *Keep* constructors can be fused, and two consecutive *Drop* constructors can be fused. Thus, *Keep* and *Drop* must alternate. Then, for greater compactness and simplicity,

these two constructors can be combined into a single constructor, say *Seg*, which carries two integer arguments *gap* and *len*. These ideas lead to representing a thinning as a *segment list*, a data structure that we describe again later on (§2.3).

In terms of asymptotic complexity, segment lists offer a potential improvement over the naive inductive representation of thinnings. The time complexity of lookup is  $O(W)$ , where  $W$  is the *weight* of the segment list, that is, the number of segments in the list, as opposed to  $O(n)$ , where  $n$  is the number of points in the codomain of the thinning. Because no segment has length zero, we have  $W \leq m \leq n$ . This is potentially interesting, but not a great improvement over the naive representation.

### 1.3 A new representation of thinnings

In an ideal world, what operations should a representation of thinnings efficiently support? The following operations seem crucial:

- (1) The trivial thinning (*identity*).  
This is analogous to the constructor *Done*.
- (2) Inserting a new diagonal segment in front of a thinning (*northeast*).  
This is analogous to the constructor *Keep*.
- (3) Inserting a new vertical gap in front of a thinning (*north*).  
This is analogous to the constructor *Drop*.
- (4) Testing whether a thinning is nonempty and inspecting its first segment (*view*).  
This is analogous to pattern matching on *Done* versus *Drop/Keep*.
- (5) Looking up the image of an index  $x$  through a thinning  $\rho$  (*get*).
- (6) Composing two thinnings (*compose*).

This paper presents an original data structure, *segment maps*, which supports the above operations as well as a few more (*west*; *invert*; *prune*) with good asymptotic time complexity. Our construction and destruction operations (*northeast*; *north*; *view*) have complexity  $O(1)$ . Lookup (*get*) has complexity  $O(\log W)$ . Composition (*compose*) has complexity  $O(W_1 + W_2)$  in the worst case. Furthermore, if one of its arguments has weight  $O(1)$  then *compose* has complexity  $O(\log W)$  where  $W$  is the weight of its other argument. In short, our approach is to retain the basic principle of segment lists and to replace the singly-linked-list structure with the structure of a skew binary random-access list (Okasaki 1999, §9.3.1).

The paper focuses on the presentation of this data structure and on an analysis of its asymptotic complexity. It does not offer an empirical evaluation of the data structure in the setting of a realistic application.

The paper is accompanied with implementations of segment maps in OCaml<sup>1</sup> and in Rocq.<sup>2</sup> The functional correctness of the Rocq implementation is machine-checked. The time complexity analyses presented in the paper are informal. Our OCaml code is virtually identical to our Rocq code, but is hand-written, that is, *not* extracted out of Rocq. This lets us use runtime assertions in OCaml, which are useful during debugging. It also helps us ensure that the code is readable and is not polluted with *fuel* parameters or other artifacts of the extraction process. Our OCaml code has been submitted to heavy random testing using Monolith (Pottier 2021). In the paper, we present just the most interesting excerpts of the Rocq code. For further details, we refer the reader to the online repositories.

<sup>1</sup><https://gitlab.inria.fr/fpottier/segmap>

<sup>2</sup><https://gitlab.inria.fr/fpottier/rocq-segmap>

## 2 Specification

We decide to let a segment map represent an increasing function of  $\mathbb{N}$  to  $\mathbb{N}$ , whose domain and codomain are *infinite*, as opposed to a thinning, which is a function of  $[0, m)$  to  $[0, n)$ , whose domain and codomain are *finite*. Naturally, we are able to represent only the functions that have finite extent in a certain sense (§2.1). This design decision leads to a slightly simpler API. The type of segment maps is not indexed with  $m$  and  $n$ . A lookup is never out of bounds. The composition of two segment maps is always well-defined. The impact of this decision seems minor: indeed, the finite-function API can be implemented on top of the total-function API, and vice-versa.

Throughout the paper, we work with integers in  $\mathbb{Z}$ , as opposed to natural numbers in  $\mathbb{N}$ , because they let us express positive and negative displacements.

### 2.1 Mathematical model

The mathematical objects that we wish to represent are *increasing functions* of the nonnegative integers to the nonnegative integers, that is, functions  $f : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$  that obey the law:

$$\forall i. \forall j. \quad 0 \leq i < j \Rightarrow 0 \leq f(i) < f(j)$$

This law implies  $\forall i. 0 \leq i \Rightarrow f(i) + 1 \leq f(i + 1)$ , which means that the function  $f$  must everywhere grow at least as fast as a diagonal function. By a *diagonal function*, we mean a function  $d : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$  such that, for some constant  $k \in \mathbb{Z}^+$ ,  $\forall i. d(i) = i + k$  holds.

Naturally, one cannot expect a data structure to be able to describe *every* increasing function of the nonnegative integers to the nonnegative integers. We restrict our attention to the functions that have finite extent. A function  $f$  has *finite extent* provided that, beyond a certain point, it is a diagonal function:

$$\exists x \in \mathbb{Z}^+. \exists k \in \mathbb{Z}^+. \forall i \in \mathbb{Z}^+. x \leq i \Rightarrow f(i) = i + k$$

The *extent* of  $f$ , written  $\text{extent}(f)$ , is the least  $x \in \mathbb{Z}^+$  such that the above property holds.

### 2.2 Visual representation

Even though we work with discrete functions, which map integers to integers, we suggest visualizing these functions in terms of diagonal line segments in the two-dimensional plane. We use the words *north*, *south*, *west*, and *east* to designate the four directions: that is, we consider these words synonyms for *up*, *down*, *left*, and *right*.

An increasing function can be graphically represented as a finite sequence of semi-open diagonal line segments, terminated with a diagonal half-line. An example appears in Figure 2. In this drawing, each of the three line segments is semi-open. The westernmost southernmost point, represented by a black rounded dot, is part of the segment; the easternmost northernmost point, represented by a white rounded dot, is not part of the segment. The three line segments are followed with a half-line, which extends all the way to infinity, as suggested by the square dots. This drawing is a graphical representation of the increasing function  $f$  defined by  $f(0) = 0$ ,  $f(1) = 2$ ,  $f(2) = 5$ ,  $f(3) = 6$ , and  $\forall i. 4 \leq i \Rightarrow f(i) = i + 4$ .

The extent of this function is 4. It is both the total width and the total height of the three line segments that appear in the graph of the function  $f$ .

The *weight* of a function is the number of line segments that appear in its graph. Therefore, the weight of the function  $f$  in Figure 2 is 3.

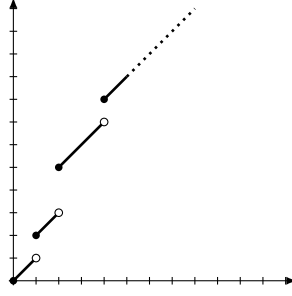


Fig. 2. An increasing function of weight 3 and extent 4

### 2.3 Segment lists

As suggested in introduction (§1), a simple data structure that represents an increasing function is the *segment list*. A segment list is a finite sequence of segments, where each segment is described by an *altitude* and a *length*. This sequence is terminated with a half-line, which is described by an *altitude*. An *altitude* is a nonnegative integer. It measures the vertical distance from the horizontal axis to the starting point of a line segment or half-line. A *length* is a positive integer. It measures the length of a diagonal segment. In other words, it measures a move towards the east and a move of equal length towards the north.

In Figure 2, the first segment is described by altitude 0 and length 1, the second segment by altitude 2 and length 1, the third segment by altitude 5 and length 2, and the half-line by altitude 8. By convention, an altitude is relative to the horizontal axis: it is not relative to the endpoint of the previous segment.

For the sake of brevity, we omit the inductive definition of segment lists. Segment lists are explicitly used in a few places in this paper, namely in our first two composition algorithms (Figure 15 and Figure 17). Furthermore, the operation *view* (§2.4), which is used in our final composition algorithm (Figure 19), can be understood as an incremental way of converting a segment map to a segment list.

### 2.4 Operations

In the following, we let the word *map* denote a data structure that represents an increasing function with finite extent. We expect an implementation of maps to provide at least the following constants and operations:

- (1) *get*. This operation returns the image of an index  $x \in \mathbb{Z}^+$  through the function represented by a map  $m$ . In other words, it computes  $m(x)$ . We refer to this operation as a *lookup*. This operation does not have a formal specification: instead, the specification of every other operation is expressed using *get*.
- (2) *identity*. This map represents the identity function. Its specification is *get identity*  $x = x$ . It can be visualized as the half-line whose starting point is  $(0, 0)$ .
- (3) *north*. This operation moves a map  $m$  towards the north by *jump* units, where *jump* can be positive or negative, so a move towards the north or towards the south is possible. Its specification is *get (north jump m)*  $x = \text{get } m \ x + \text{jump}$ . The parameter *jump* must obey the constraint  $0 \leq \text{get } m \ 0 + \text{jump}$ . This implies  $\forall x \in \mathbb{Z}^+ . 0 \leq \text{get } (\text{north jump } m) \ x$ , which means that the map cannot be moved below the horizontal axis. Figure 3 offers

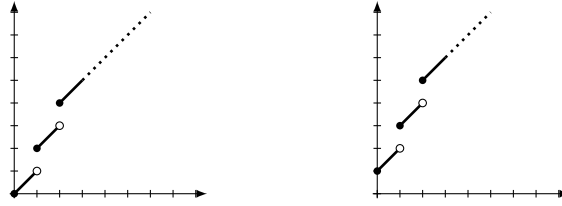


Fig. 3. A move towards the north by 1 unit

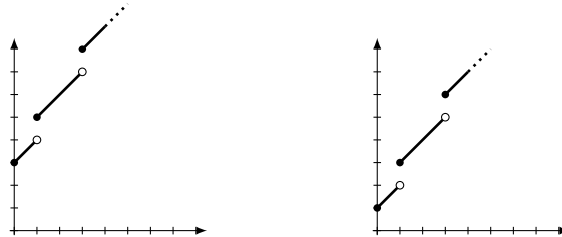


Fig. 4. A move towards the south by 2 units

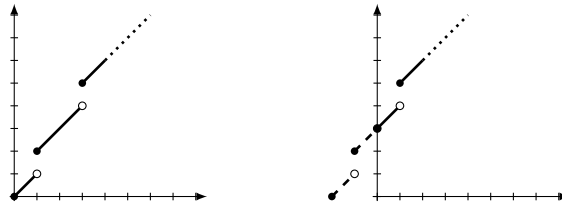


Fig. 5. A move towards the west by 2 units

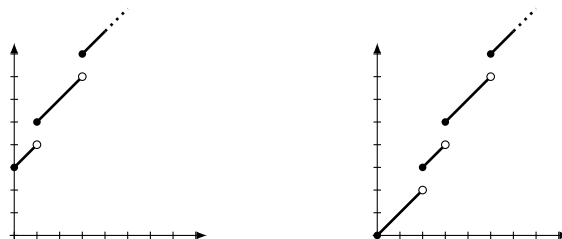


Fig. 6. A move towards the east by 2 units and the insertion of a new segment, without fusion

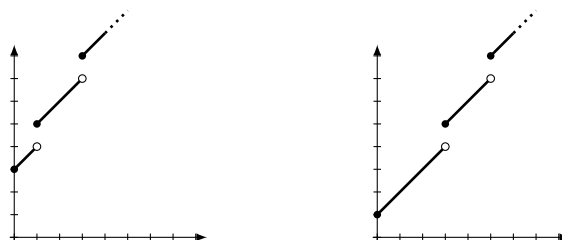


Fig. 7. A move towards the east by 2 units and the insertion of a new segment, with fusion

an example where *jump* is 1. The left-hand side of the figure shows the map before the move; the right-hand side shows the map that is obtained after the move. Figure 4 offers an example where *jump* is -2.

- (4) *west*. This operation moves a map  $m$  towards the west by a nonnegative amount  $\delta$ . Its specification is  $\text{get } (\text{west } \delta m) x = \text{get } m (\delta + x)$ . Figure 5 offers an example where  $\delta$  is 2. This operation destroys information: the part of the map that moves to the west of the vertical axis is lost. In Figure 5, the part that is destroyed is drawn in dashed lines. In this example, the first segment of the original map disappears entirely. The second segment is shortened and becomes the first segment of the new map.
- (5) *east*. This operation moves a map  $m$  towards the east by a nonnegative amount  $\delta$ . At the same time, it creates a new segment, whose length is  $\delta$  and whose starting *altitude* is specified by the user. This new segment is inserted in front of the map  $m$  and is possibly fused with its first segment. The specification of this operation is  $\text{get } (\text{east } \text{altitude } \delta m) x = \text{if } x < \delta \text{ then } \text{altitude} + x \text{ else } \text{get } m (x - \delta)$ . Figure 6 offers an example where *altitude* is 0 and  $\delta$  is 2. No fusion takes place. Figure 7 offers a second example, where *altitude* is 1 and  $\delta$  is 2. There, the new segment is fused with the pre-existing first segment, resulting in a segment of length 3. The parameters of this operation must obey the constraint  $\text{altitude} + \delta \leq \text{get } m 0$ , which guarantees that the result represents an increasing function. If  $\text{altitude} + \delta = \text{get } m 0$  holds then fusion takes place, as in Figure 7. If  $\text{altitude} + \delta < \text{get } m 0$  holds then no fusion takes place, as in Figure 6.
- (6) *invert*. Given a map  $m$  and an altitude  $y$ , this operation returns the least index  $x \in \mathbb{Z}^+$  such that  $y \leq \text{get } m x$  holds. We refer to this operation as a *reverse lookup*.
- (7) *compose*. This operation constructs the sequential composition of two maps  $m_1$  and  $m_2$ . Its specification is  $\text{get } (\text{compose } m_1 m_2) x = \text{get } m_2 (\text{get } m_1 x)$ .
- (8) *equal*. This operation determines whether two maps  $m_1$  and  $m_2$  represent the same function. Its specification is  $\text{equal } m_1 m_2 = \text{true} \iff \forall x. \text{get } m_1 x = \text{get } m_2 x$ .
- (9) *view*. This operation accepts a map  $m$  and returns a description of this map as either just a half-line (*VId*) or the concatenation of one segment and a map (*VSeg*). Its specification is as follows. If  $\text{view } m$  returns *VId gap*, then  $\forall x. \text{get } m x = \text{gap} + x$  holds. If  $\text{view } m$  returns *VSeg gap len m'* then we have  $\forall x. \text{get } m x = \text{if } x < \text{len} \text{ then } \text{gap} + x \text{ else } \text{get } m' (x - \text{len})$ .

One important operation can be implemented using *north* and *east*. *northeast* moves a map  $m$  towards the north by  $\delta$  units and towards the east by  $\delta$  units, creating a new segment whose origin is  $(0, 0)$  and whose length is  $\delta$ . Its specification is  $\text{get } (\text{northeast } \delta m) x = \text{if } x < \delta \text{ then } x \text{ else } \text{get } m (x - \delta) + \delta$ . In an application to terms whose variables are represented as de Bruijn indices, *northeast delta* is typically used to move a renaming under  $\delta$  binders, extending both the domain and the codomain of this renaming.

### 3 Segment maps

Our segment maps are based on Okasaki's skew binary random-access lists, whose structure and properties we briefly recall (§3.1). Then, we present the structure of segment maps (§3.2), their meaning (§3.3), and the well-formedness properties that they must obey (§3.4). We conclude this section with a discussion of a few alternative designs (§3.5).

### 3.1 Random access lists

A skew binary random-access list (Okasaki 1999, §9.3.1) represents a sequence of elements. This data structure supports inserting an element in front of a sequence (*cons*) and removing the first element of a sequence (*tail*) in time  $O(1)$ . Furthermore, it offers fast random access (*lookup*): the  $i$ -th element of a sequence can be found in time  $O(\min(i, \log n))$ , where  $n$  is the number of elements of the sequence. We remark that it also supports removing the first  $i$  elements of a sequence (*drop*) in time  $O(\min(i, \log n))$ . This is exploited in our implementation of *west*.

The structure of a skew binary random-access list involves two inductive data types, namely *trees* and *random-access lists*. A tree is either a leaf, which carries one element, or a node, which carries one element and two subtrees. By convention, every tree is *perfect*: all of its leaves lie at the same depth. This implies that the weight of a tree (the number of elements that it contains) must be a power of two, minus one. A random-access list is a list of trees. We refer to this list as the *spine*. The weights of the trees along the spine form an increasing sequence, possibly with one exception: the first two trees may have the same weight.

### 3.2 Definition of segment maps

As the basis of our segment maps, we use Okasaki’s skew binary random-access lists. In short, a segment map is a sequence of diagonal segments, and we organize this sequence as a random-access list. A definition of segment maps in Rocq (The Rocq team 2026) appears in Figure 8.

Rocq’s **Implicit Type** command lets us conventionally associate a variable name with a type, removing the need for many type annotations. For example, we set the convention that the variables *len* and *gap* have type  $\mathbb{Z}$ , the type of integers. Thus, in the definition of the constructor *Leaf*, Rocq understands that this constructor carries two arguments whose type is  $\mathbb{Z}$ , and we understand that these arguments are usually named *len* and *gap*.

A *segment* is described by a jump towards the north by *gap* units and by its length *len*. A segment is oriented towards the northeast. Thus, its starting point is  $(0, \text{gap})$  and its endpoint is  $(\text{len}, \text{gap} + \text{len})$ .

A *tree* is either a leaf, which describes one segment, or a node, which describes a succession of one segment and two subtrees. The constructor *Leaf* carries two fields, *len* and *gap*, which describe one segment. The constructor *Node* carries five fields, namely  $w$ , *gap*, *len*,  $t_1$ , and  $t_2$ . Together, *gap* and *len* describe the first segment. The subtrees  $t_1$  and  $t_2$  describe more segments. The field  $w$  stores the total width of this tree, that is to say, *len* plus the width of  $t_1$  plus the width of  $t_2$ . In our interpretation of the constructor *Node*, we adopt the convention that the parameter *gap* plays a double role: not only is it part of the description of the first segment, but it also represents a vertical displacement that applies to the subtrees  $t_1$  and  $t_2$ . This lets us move an entire tree towards the north in constant time by adjusting the *gap* field at its root. One way of describing how *Node* must be interpreted is: first, move the origin to  $(0, \text{gap})$ ; then, side by side, on a common base line, draw a diagonal segment of length *len*, the tree  $t_1$ , and the tree  $t_2$ . Thus the origin of the first subtree is the point  $(\text{len}, \text{gap})$  and the origin of the second subtree is the point  $(\text{len} + \text{width } t_1, \text{gap})$ . These conventions are depicted in Figure 9.

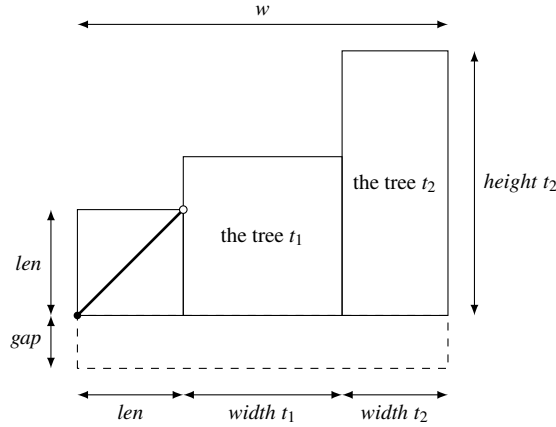
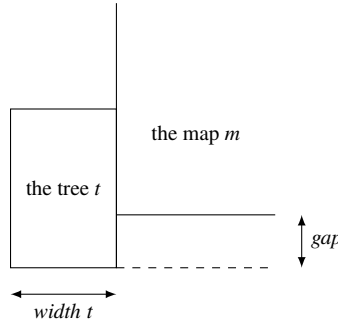
A *map* is either a half-line or the juxtaposition of a tree and a map on a common base line. The constructor *Id* represents a half-line, whose starting point is  $(0, \text{gap})$ . The constructor *Cons*

```

1 Implicit Type gap jump : Z.      (* a vertical gap or jump *)
2 Implicit Type len      : Z.      (* the length of a diagonal segment *)
3 Implicit Type w        : Z.      (* the horizontal width of a tree *)
4 Implicit Type wghm     : Z.      (* the weight of a map *)
5 Implicit Type wght     : Z.      (* the weight of a tree *)
6
7 Inductive tree :=
8   | Leaf len gap
9   | Node w gap len (t1 t2 : tree).
10
11 Implicit Type t : tree.
12
13 Inductive map :=
14   | Id gap
15   | Cons wghm wght t gap (m : map).
16
17 Implicit Type m : map.

```

Fig. 8. Definition of segment maps and conventional metavariables

Fig. 9. Graphical interpretation of the constructor  $\text{Node } w \text{ gap } len \ t_1 \ t_2$ Fig. 10. Graphical interpretation of the constructor  $\text{Cons } wghm \ wght \ t \ \text{gap } m$

represents the horizontal juxtaposition of a tree  $t$  and a map  $m$ . The field *wghm* stores the combined *weight* of the tree  $t$  and the map  $m$ , that is, the number of segments that  $t$  and  $m$  contain. The field *wght* stores the *weight* of the tree  $t$ . This is redundant, as *wght* must be equal to *wghm* – *weight m*. Therefore, if desired, this field could be removed. The field *gap* is a delayed displacement towards the north, which must be applied to the map  $m$ . In other words, the origin of the map  $m$  is the point (*width t*, *gap*). The *gap* field lets us move an entire map towards the north in constant time. These conventions are depicted in Figure 10.

### 3.3 Lookup in a segment map

To unambiguously explain which function of  $\mathbb{Z}^+$  to  $\mathbb{Z}^+$  is represented by a tree or by a map, we present the lookup functions *tree-get* and *get*. They are shown in Figure 11, along with the auxiliary function *width*, which (in constant time) returns the width of a tree. In the call *tree-get t x*, we normally require the index  $x$  to fall within the horizontal extent of the tree; this is expressed by the inequalities  $0 \leq x < \text{width } t$ . As an exception to this rule, in our logical specifications and statements, we sometimes use the expression *tree-get t (width t)*, which (one can prove) is synonymous with *height t*. The *height* of a tree  $t$  is the altitude of the endpoint of the last segment of the tree  $t$ . For the sake of completeness, the definition of the function *height* is shown in Figure 11; yet this function is not used at runtime in our code. In the call *get m x*, we require  $0 \leq x$ .

For simplicity, the definitions of *tree-get* and *get* in Figure 11 use non-tail-calls. By carrying an accumulator, it is possible to give tail-recursive definitions of these functions, which are semantically equivalent, more difficult to read, and faster in practice. In its tail-recursive formulation, *get* is just a loop that advances along the spine until it finds the desired tree, and *tree-get* is just a loop that descends into the tree until it finds the desired node or leaf.

The worst-case time complexity of *get m x* is  $O(\min(1 + s, \log W))$  where  $s$  is the index in the sequence of the segment that contains  $x$  and where  $W$  is the weight of the map  $m$ . Because every segment has nonzero length,  $s \leq x$  holds. We write  $O(1 + s)$ , as opposed to just  $O(s)$ , because *get* has nonzero cost even when  $s$  is zero. Writing  $O(s)$  is considered acceptable informal practice but could lead us to a mistake in the complexity analysis of more complex operations, such as *west* and *clip*. We prefer to avoid this pitfall.

The complexity analysis of *get* is due to Okasaki (1999, §9.3.1). The bound  $O(\log W)$  stems from the fact that the spine has length  $O(\log W)$  and every tree has depth  $O(\log W)$ . The bound  $O(1 + s)$  stems from the fact that each recursive call dismisses at least one segment.

### 3.4 Well-formedness of segment maps

The predicates *tree-check* and *tree-minimal*, defined in Figure 12, express the invariants that we impose on trees. The predicate *tree-check* expresses the following requirements:

- (1) Every segment has nonnegative *gap* and positive *len* (lines 10 and 14).
  - (2) Every node carries two subtrees of equal weight (line 12).
- This implies that every tree is perfect.
- (3) Every node correctly records its total width (line 13).

The parameter *jump* in *tree-check jump t* is a vertical displacement that must be applied to the tree  $t$  before its well-formedness is checked. This parameter is increased in the recursive calls (line 15), reflecting the fact that the displacement *gap* applies to the subtrees  $t_1$  and  $t_2$ .

The predicate *tree-minimal* expresses the following requirements:

```

1  Definition width t :=
2    match t with
3    | Leaf w _      => w
4    | Node w _ _ _ => w
5    end.
6
7  Fixpoint tree_get t x :=
8    match t with
9    | Leaf len gap      => gap + x
10   | Node w gap len t1 t2 => (* See Figure 9: *)
11       if x <? len then gap + x
12       else if x - len <? width t1 then gap + tree_get t1 (x - len)
13       else gap + tree_get t2 (x - len - width t1)
14   end.
15
16  Fixpoint get m x :=
17    match m with
18    | Id gap => gap + x
19    | Cons wghm wght t gap m => (* See Figure 10: *)
20       if x <? width t then tree_get t x
21       else gap + get m (x - width t)
22   end.
23
24  Fixpoint height t := (* provided as documentation; unused *)
25    match t with
26    | Leaf len gap      => gap + len
27    | Node _ gap _ _ t2 => gap + height t2
28   end.

```

Fig. 11. Lookup in a segment map

- (1) Two consecutive segments can never be fused to form a single segment.
- (2) The tree represents an increasing function.

These two requirements are in fact closely related. They can be reformulated as follows:

- (1) Between two consecutive segments, there must be a nonzero jump;
- (2) Between two consecutive segments, there must be a positive jump.

This formulation makes it clear that the second requirement implies the first one. Therefore it suffices to express just the second requirement. At every node, we require  $len < tree\_get\ t_1\ 0$  (line 23), which means that there must be a positive jump from the endpoint of the segment carried by this node to the starting point of the first segment of the subtree  $t_1$ . Similarly, we require  $tree\_get\ t_1\ (width\ t_1) < tree\_get\ t_2\ 0$  (line 24), which means that there must be a positive jump from the endpoint of the last segment of the tree  $t_1$  to the starting point of the first segment of the tree  $t_2$ .

The predicates *check* and *minimal*, shown in Figure 13, play similar roles at the level of maps. The predicate *check* imposes the following requirements:

- (1) The final half-line must start at a nonnegative altitude (line 12).

```

1  Fixpoint tree_weight t :=
2    match t with
3    | Leaf _ _      => 1
4    | Node _ _ _ t1 t2 => 1 + tree_weight t1 + tree_weight t2
5    end.
6
7  Fixpoint tree_check jump t :=
8    match t with
9    | Leaf len gap =>
10      0 ≤ jump + gap ∧ 0 < len
11    | Node w gap len t1 t2 =>
12      tree_weight t1 = tree_weight t2 ∧
13      w = len + width t1 + width t2 ∧
14      0 ≤ jump + gap ∧ 0 < len ∧
15      tree_check (jump + gap) t1 ∧ tree_check (jump + gap) t2
16    end.
17
18  Fixpoint tree_minimal t :=
19    match t with
20    | Leaf len gap =>
21      True
22    | Node w gap len t1 t2 =>
23      len < tree_get t1 0 ∧
24      tree_get t1 (width t1) < tree_get t2 0 ∧
25      tree_minimal t1 ∧ tree_minimal t2
26    end.

```

Fig. 12. Well-formedness of a tree

- (2) At every *Cons* constructor, the weight of the entire map and the weight of the tree  $t$  are correctly recorded (lines 14 and 15).
- (3) At every *Cons* constructor, the weight of the tree  $t$  is less than the weight of the next tree along the spine, unless  $t$  is the front tree, in which case its weight may be equal to the weight of the next tree (line 16). This is expressed via the auxiliary predicate *can-prepend*.

The parameter *front* determines whether we are presently looking at the front of the spine. The parameter *jump* is a vertical displacement that must be applied to the map  $m$  before its well-formedness is checked. In the end, a map is well-formed if *check true 0 m* holds.

The predicate *minimal* is analogous to *tree-minimal*. At every *Cons* constructor, it requires the existence of a positive jump between the endpoint of the tree  $t$  and the starting point of the map  $m$ .

Although *check* and *minimal* could be merged into a single predicate, it is convenient in some lemmas to require or to establish just one of them.

### 3.5 Alternative designs

A possible weakness of our design is that segment maps do *not* have canonical equality: that is, two maps that represent the same mathematical function are *not* necessarily identical.

```

1  Implicit Type front : bool.
2
3  Definition can_prepend front wght m :=
4    match m with
5      | Id _           => True
6      | Cons _ wght' _ _ => wght ≤ wght' ∧ if front then True else wght < wght'
7    end.
8
9  Fixpoint check front jump m :=
10    match m with
11      | Id gap =>
12        0 ≤ jump + gap
13      | Cons wghm wght t gap m =>
14        wghm = tree_weight t + weight m ∧
15        wght = tree_weight t ∧
16        can_prepend front wght m ∧
17        tree_check jump t ∧ check false (jump + gap) m
18    end.
19
20 Fixpoint minimal m :=
21   match m with
22     | Id gap =>
23       True
24     | Cons wghm wght t gap m =>
25       tree_get t (width t) < gap + get m 0 ∧
26       tree_minimal t ∧ minimal m
27   end.

```

Fig. 13. Well-formedness of a segment map

This is due to the delayed displacement *gap* carried by the constructor *Cons*. In our OCaml and Rocq implementations in our proofs of functional correctness, the absence of canonical equality is not a problem. Our implementation offers an equality test (*equal*) whose worst-case time complexity is linear. However, a user who wishes to use our Rocq library in a verified program may find that the lack of canonical equality is inconvenient.

To remedy this potential problem, we believe that one could remove the *gap* field carried by the constructor *Cons wghm wght t gap m*. One would then change the interpretation of this constructor: by convention, the baseline of the map *m* would be *height t*. Therefore, to move both *t* and *m* towards the north by *gap* units, it would suffice to move just *t*, and this can be done in time  $O(1)$ . To avoid slowing down *get*, it would be necessary to have access to the height of a tree in time  $O(1)$ . For this purpose, every *Node* constructor should carry one more field, storing the height of this subtree. Because there is a linear number of *Node* constructors, as opposed to a logarithmic number of *Cons* constructors, this alternative design would consume slightly more space.

If *height t* can be obtained in time  $O(1)$  then *invert* can take advantage of it and can terminate faster in some cases. This does not change its worst-case time complexity.

An implementor who is concerned with maximum performance in space and time may wish to save memory by packing several integers together in just one machine word, at the cost of representing a gap, a width, or a length with fewer than 63 bits. This would allow saving one word in every *Leaf* object and one or two words in every *Node* object. This could also allow every *Node* object to store *height* information at no extra cost in space. In *Cons* objects, some space can also be saved, but this seems less interesting, as there is only a logarithmic number of them.

The reader may wonder whether one could implement segment maps *on top of* Okasaki's skew binary random-access lists, while maintaining an abstraction barrier between the two data structures. This seems unlikely. One would need to generalize the API and the implementation of random-access lists so much that, in the end, almost all of the code would be placed in the random-access-list layer, leaving almost none in the segment-map layer.

## 4 Operations on segment maps

We now review how the main operations on segment maps are implemented. Most of our attention is focused on the composition algorithm (§4.6), on its auxiliary operations *west* (§4.3) and *clip* (§4.4), and on an analysis of the time complexity of these operations. Most of the code is omitted. We provide code only for the auxiliary function *view* and for three variants of *compose*.

### 4.1 North

A move towards the north is a matter of adjusting one or several *gap* fields. Its worst-case time complexity is  $O(1)$ .

### 4.2 East

A move towards the east is implemented by two functions. An auxiliary function, *seg*, deals with the case where no fusion takes place (Figure 6). Then, one new segment must be prepended to the sequence of segments. This operation is analogous to Okasaki's *cons*. If the first two trees in the spine have equal weights then the new segment is combined with these two trees to form a larger tree (*Node*). Otherwise the new segment, alone, forms a new tree (*Leaf*). The worst-case time complexity of this operation is  $O(1)$ . The function *seg* is so named because it prepends one segment in front of a sequence of segments: it behaves like a constructor of a segment list (§2.3).

The main function, *east*, determines whether the new segment must be fused with the first segment of the current map, as is the case in Figure 7. If so, the first segment of the current map is updated. Otherwise, *seg* is called. In either case, the time complexity of this operation is  $O(1)$ .

### 4.3 West

The call *west delta m* moves the map *m* towards the west by *delta* units (Figure 5). One can think about it as a *cut*: this operation cuts the map *m* at index *delta*, discards the segments that lie west of the cut, and keeps those that lie east of the cut. If a segment lies across the cut, it is split at the cut into two segments, and only the second segment is retained.

The implementation of this operation involves two functions, *tree-west* and *west*. These functions are similar to *tree-get* and *get* (Figure 11): they perform a search for index *delta* in

the tree or map, and they are implemented as a tail-recursive descent into the tree or map. The reader may wonder, what is the type of *tree-west*? Cutting a perfect tree at an arbitrary index *delta* does not produce a perfect tree, so one cannot expect *tree-west* to return a tree. Instead, *tree-west* expects not only a tree *t* (the first tree of a spine) but also a map *m* (the rest of this spine). Cutting the tree *t* at index *delta* produces a sequence of subtrees of various sizes, which are prepended in front of the map *m* via the constructor *Cons*, so *tree-west* returns a map. The weights of the trees that are thus prepended in front of *m* are less than the weight of *t* and form an increasing sequence. Therefore this use of *Cons* produces a well-formed random-access list. This is analogous to a *drop* operation on a random-access list.

Because they implement the same search pattern, *get* and *west* have the same time complexity. Thus, the time complexity of *west delta m* is  $O(\min(1 + s, \log W))$  where *s* is the index of the segment that contains index *delta* and where *W* is the weight of the map *m*.

We remark that *s* is in fact equal to the weight decrease *weight m* – *weight m'*, where *m'* is the map returned by *west*. In other words, the cost of *west* is (at worst, and up to a constant factor) one plus the number of segments that are thrown away.

#### Complexity of a sequence of cuts

The previous remark leads to an important result about the time complexity of a sequence of calls to *west*. Suppose that, beginning with a map *m*, one performs a sequence of *k* *west* operations, where each operation is applied to the result of the previous operation. We refer to such a sequence as a *sequence of cuts*. Then, the cost of each operation in the sequence is bounded by one plus the number of segments that are thrown away. Therefore the total cost of this sequence of operations is  $O(k + W)$  where *W* is the weight of the map *m*. Furthermore, because the cost of each operation in the sequence is also  $O(\log W)$ , the cost of the sequence is also  $O(k \log W)$ . Therefore it is  $O(\min(k + W, k \log W))$ . We exploit this fact in our analysis of the time complexity of *compose* (§4.6).

### 4.4 Clip

The auxiliary operation *clip* is related to *west*, but has the opposite effect. Whereas *west delta m* cuts the map *m* at index *delta* and discards the material that lies west of the cut, *clip delta m accu* cuts the map *m* at index *delta* and throws away the material that lies east of the cut. The segments and subtrees that lie west of the cut are retained and glued (concatenated) in front of the accumulator *accu*, which itself is a map. The specification of *clip* is *get (clip delta m accu) x = if x < delta then get m x else get accu (x – delta)*.

The implementation of *clip* and its auxiliary function *tree-clip* is analogous to that of *west* and *tree-west*. This time, though, to concatenate the retained segments and subtrees in front of the map *accu*, one cannot use the constructor *Cons*. Indeed, the weights of these subtrees do not necessarily form an increasing sequence; furthermore, there is no relationship between the weights of these trees and the weight of the first tree in the map *accu*. Therefore, instead of *Cons*, we use an auxiliary function, *concat*.

In short, *concat* is a gluing (concatenation) operation: *concat t accu* inserts the segments of the tree *t* in front of the map *accu*. A simple implementation of *concat* iterates over *t* and inserts each segment in turn into the map *accu* using the auxiliary function *seg* (§4.2). Its time complexity is  $O(W)$  where *W* is the weight of the tree *t*.

Provided one keeps track of the weight of the tree  $t$ , one can propose a slightly better implementation of *concat*. Based on up to two comparisons of weights, sometimes one determines that it is permitted to use the constructor *Cons* to insert the entire tree  $t$  at once in front of the map *accu*. When this is not permitted, the tree  $t$  must be decomposed into one segment and zero or two subtrees, which are inserted into the map *accu* using *seg* and *concat*. We expect this improved implementation of *concat* to be slightly faster in practice than the naive one based on iterating *seg*. Nevertheless, its worst-case time complexity is still  $O(W)$  where  $W$  is the weight of the tree  $t$ .

Because *concat* is more expensive than *Cons*, *clip* is more expensive than *west*. Its cost is the cost of cutting plus the cost of gluing. Thus, its time complexity can be described as follows. The cost of computing  $\text{accu}' = \text{clip } \text{delta } m \text{ accu}$  is the cost of  $\text{west } \text{delta } m$  plus the difference in weight between  $\text{accu}'$  and *accu*.

### Complexity of a sequence of cuts and clips

The cost of a sequence of *clip* operations plays a role in the analysis of the time complexity of *compose* (§4.6). Suppose that, beginning with a map  $m$ , one performs a sequence of cuts, as described earlier (§4.3), that is, a sequence of  $k$  successive *west* operations. As intermediate results, this produces a sequence of maps  $m_0, m_1, \dots, m_k$ , where every  $m_i$  can be written under the form  $\text{west } \text{delta}_i m$ , for some increasing sequence of displacements  $\text{delta}_0 \leq \text{delta}_1 \leq \dots \leq \text{delta}_k$ , where  $\text{delta}_0$  is zero. Now, suppose that, to each intermediate map  $m_i$ , one applies a *clip* operation  $\text{clip } \text{len}_i m_i$ , so as to copy a slice of this map. Suppose furthermore that the lengths  $\text{len}_i$  are chosen in such a way that the slices are pairwise disjoint. (That is, suppose  $\text{len}_i \leq \text{delta}_{i+1} - \text{delta}_i$  holds.) Therefore every segment of the map  $m$  is fully contained in at most one slice. What is the total cost of these calls to *clip*?

The cost of  $\text{clip } \text{len}_i m_i$  is the cost of cutting, that is, searching for index  $\text{len}_i$  in the map  $m_i$ , plus the cost of gluing, that is, concatenating the slice thus obtained in front of the accumulator.

The cost of cutting the  $i$ -th slice is bounded, up to a constant factor, by one plus the number of segments of the map  $m$  that have a nonempty intersection with this slice. Thus it is also bounded, up to a constant factor, by one plus the number of segments of the map  $m$  that are fully contained in this slice. (The difference between these bounds is at most two, and none of them can be zero, so they differ by a constant factor.) Therefore, recalling that no segment of the map  $m$  can be fully contained in two distinct slices, the total cost of cutting is  $O(k + W)$ , where  $W$  is the weight of the map  $m$ . Clearly it is also  $O(k \log W)$ , so it is  $O(\min(k + W, k \log W))$ .

The cost of gluing (copying) the  $i$ -th slice is bounded, up to a constant factor, by one plus the weight of this slice, that is, one plus the number of segments in this slice. In fact, via an accounting trick, one can give a better bound. As part of the cost of cutting the  $i$ -th slice, we have already accounted for the fact that the  $i$ -th call to *clip* costs one unit of time. Therefore, in our analysis of the cost of gluing, we may remove the term “one plus” and consider that the cost of cutting the  $i$ -th slice is just the weight of this slice. Even better, we may consider that the cost of cutting the  $i$ -th slice is the *restricted weight* of this slice, where we define the restricted weight of a slice as the number of segments in this slice that end at or below index  $E$ , where  $E$  is the extent of the map  $m$ . The call  $\text{clip } \text{len}_i (\text{west } \text{delta}_i m)$  can produce at most one segment that begins at index  $E$  or higher, and no segment that

straddles index  $E$ . Therefore the restricted weight of a slice and the weight of this slice differ by at most one. Therefore this accounting scheme is sound.

We are now in a position to conclude. The cost of gluing is  $O(\hat{S})$ , where  $\hat{S}$  denotes the total restricted weight of the slices that are taken.  $\hat{S}$  can be bounded in two ways. First, because the restricted weight of a slice is bounded by its weight,  $\hat{S}$  is bounded by  $S$ , the total weight of the slices that are taken. Second, because the restricted weight of a slice is bounded by the width of its intersection with the interval  $[0, E)$ , where  $E$  is the extent of the map  $m$ ,  $\hat{S}$  is bounded by  $E$ .

In summary, in a sequence of cuts and clips, the total cost of the calls to *clip* is the cost of cutting, namely  $O(\min(k + W, k \log W))$ , plus the cost of gluing, namely  $O(\min(S, E))$ .

#### 4.5 Invert

The call *invert m y* computes the least index  $x$  such that  $y \leq \text{get } m x$  holds. Because every map  $m$  represents an increasing function, *invert* can be implemented as a dichotomic search. Its time complexity is  $O(\min(1 + s, \log W))$  where  $s$  is the index of the segment where  $x$  lies and  $W$  is the weight of the map  $m$ .

#### 4.6 Compose

Composition is arguably the most interesting and most elaborate operation on segment maps. There seem to be two natural ways of approaching the composition of two maps  $m_1$  and  $m_2$ . One approach is to iterate over the segments of  $m_1$ . This determines vertical slices in  $m_2$ , which are cut and glued using *west* and *clip*. The second approach is to iterate over the segments of  $m_2$ . This determines horizontal slices in  $m_1$ . Using *invert*, these slices are converted into vertical slices in  $m_1$ , which are vertically slid, cut and glued using *north*, *west* and *clip*. We present these two algorithms and analyze their time complexity. Then we propose a hybrid algorithm that is always at least as fast as either algorithm.

##### 4.6.1 Composition via iterating on $m_1$ and slicing $m_2$

**Illustration.** The steps of our first composition algorithm are illustrated in Figure 14. In this example, the maps that must be composed are  $m_1$ , shown in Figure 14a, and  $m_2$ , shown in Figure 14b.

In Figure 14a, as a background, we show a series of colored horizontal slices. A green slice corresponds to a segment or a half-line in the map  $m_1$ . A red slice corresponds to a vertical gap in the map  $m_1$ . Therefore, the union of the green slices corresponds to the codomain of the increasing function  $m_1$ .

In Figure 14c, we transform the horizontal slices determined by  $m_1$  into vertical slices. To do so, we apply the symmetry whose axis is the diagonal line ( $y = x$ ). On top of these colored vertical slices, we show the map  $m_2$  again. The segments and half-line that form the map  $m_2$  are cut at a boundary between two slices; new endpoints are created there. Inside a red slice, the graph of  $m_2$  is drawn in dashed lines.

It is now hopefully clear that the composition of  $m_1$  and  $m_2$  is constructed by discarding the red slices in Figure 14c and by concatenating the green slices. The result appears in Figure 14d.

**Code.** The code of this composition algorithm appears in Figure 15. Because we need to iterate over the segments of the map  $m_1$ , we convert this map to a *segment list* (line 14)

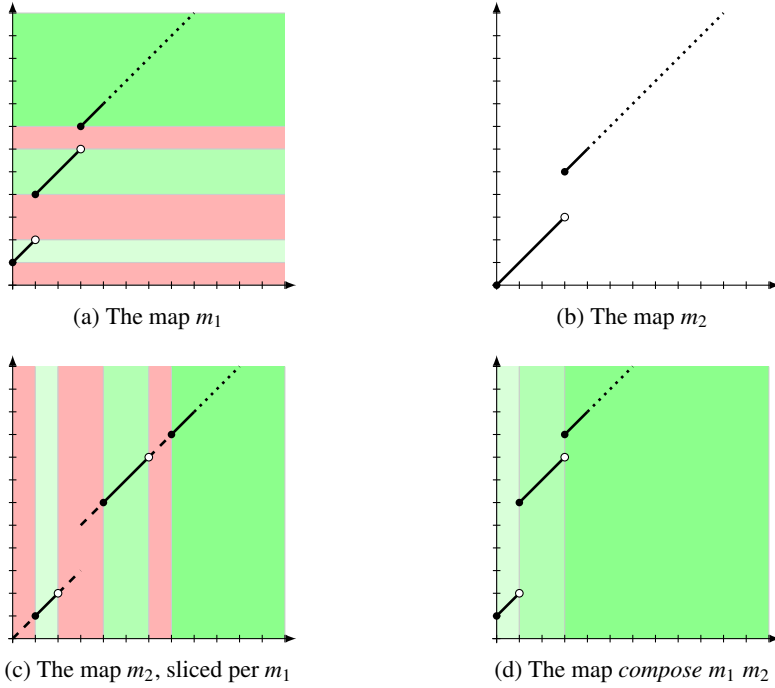


Fig. 14. The steps of the first composition algorithm

```

1 Fixpoint compose_1_aux gap1 segs1 m2 :=
2   match segs1 with
3   | SId gap =>
4     let gap := gap1 + gap in
5     west gap m2
6   | Seg gap len segs1 =>
7     let gap := gap1 + gap in
8     let m2 := west gap m2 in
9     clip len m2
10    (compose_1_aux (gap1 - gap) segs1 m2)
11  end.
12
13 Definition compose_1 m1 m2 :=
14  compose_1_aux 0 (to_seglist m1) m2.

```

Fig. 15. The first composition algorithm

using the auxiliary function *to-seglist* (not shown). The constructors of the inductive type of segment lists (also not shown) are *SId* and *Seg*.

The recursive function *compose-1-aux* expects a gap  $gap_1$ , a segment list  $segs_1$ , and a map  $m_2$ . The parameter  $gap_1$  represents a northern move that must be applied to  $segs_1$ . We introduce this parameter because this lets us avoid the need for an operation *north*

on segment lists; such an operation would be inefficient. To understand *compose-I-aux*, we suggest focusing on the situation where  $gap_1$  is zero.

The function *compose-I-aux* performs a case analysis on the segment list. Let us examine the case where this segment list begins with a segment described by  $gap$  and  $len$ , followed with a sublist  $segs_1$  (line 6). In the visual language of Figure 14a, this means that the bottommost two colored slices are a red slice whose height is  $gap$  and a green slice whose height is  $len$ . In terms of Figure 14c, now, this means that we must first discard a vertical slice of  $m_2$  whose width is  $gap$ , then keep a vertical slice of  $m_2$  whose width is  $len$ . These two operations are performed by *west gap* and *clip len* (lines 8 and 9). At this point, there remains to continue, via a recursive call to *compose-I-aux*. A technical point could easily be overlooked: because we have moved the map  $m_2$  towards the west by  $gap$  units, we must accordingly move  $segs_1$  towards the south by  $gap$  units. This explains why we request a move of  $-gap$  units towards the north as part of the parameters of the recursive call (line 10).

**Complexity.** Let us write  $E_1$  and  $E_2$  for the extents of the maps  $m_1$  and  $m_2$ ,  $W_1$  and  $W_2$  for the weights of the maps  $m_1$  and  $m_2$ , and  $W'$  for the weight of the result of the composition, that is, the weight of the map *compose*  $m_1 m_2$ .

One can bound the weight  $W'$  as follows:  $W_1 \leq W' \leq W_1 + W_2$ . Intuitively,  $W_1 \leq W'$  holds because every discontinuity in the graph of the function  $m_1$  must cause a discontinuity in the graph of the composite function.  $W' \leq W_1 + W_2$  holds because every discontinuity in the graph of the composite function must be caused by a discontinuity in  $m_1$  or in  $m_2$ .

The cost of converting  $m_1$  to a segment list, and of iterating on this segment list, is  $O(W_1)$ .

The calls to *west* performed by the composition algorithm form a sequence of cuts (§4.3): that is, each *west* operation is applied to the result of the previous *west* operation. The number of operations in this sequence is  $W_1$ , and the starting point of this sequence is the map  $m_2$ , whose weight is  $W_2$ . According to our earlier analysis of the time complexity of a sequence of cuts (§4.3), the total cost of these calls is  $O(\min(W_1 + W_2, W_1 \log W_2))$ .

There remains to assess the cost of the calls to *clip*. The manner in which *west* and *clip* are used forms a sequence of cuts and clips, as studied earlier (§4.4). Therefore the total cost of the calls to *clip* is the cost of cutting, namely  $O(\min(W_1 + W_2, W_1 \log W_2))$ , plus the cost of gluing, namely  $O(\min(S, E_2))$ , where  $S$  is the total weight of the slices that are taken. Now,  $S$  is bounded both by  $W'$ , the weight of the result, and by  $E_1$ , the extent of the map  $m_1$ . Therefore the cost of gluing is  $O(\min(W', E_1, E_2))$ .

In summary, the time complexity of this composition algorithm is

$$O \left( \begin{array}{l} \min(W_1 + W_2, W_1 \log W_2) \\ + \min(W', E_1, E_2) \end{array} \right)$$

This bound has two important consequences:

- (1) In all situations, the algorithm has linear time complexity in the weights of the maps, that is,  $O(W_1 + W_2)$ . This stems from the inequality  $W' \leq W_1 + W_2$ .
- (2) If  $E_1$  is considered a constant then, because of the inequality  $W_1 \leq E_1$ , the weight  $W_1$  can also be considered a constant. Then one finds that the algorithm has complexity  $O(\log(W_2))$ .

We summarize these two results as follows: this composition algorithm behaves reasonably well in all circumstances; and if  $m_1$  has small extent then it is very efficient.

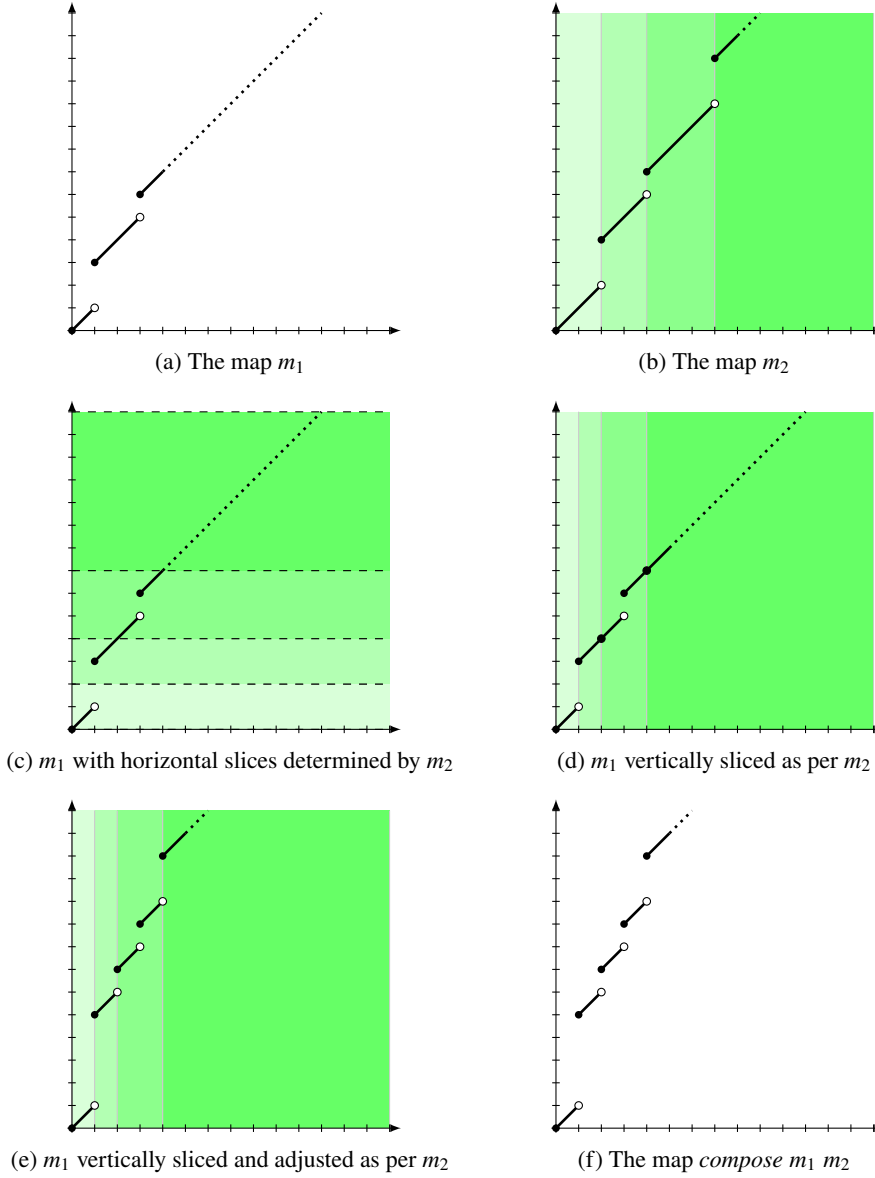


Fig. 16. The steps of the second composition algorithm

#### 4.6.2 Composition via iterating on $m_2$ and slicing $m_1$

**Illustration.** The steps of our second composition algorithm are illustrated in Figure 16. In this example, the maps that must be composed are  $m_1$ , shown in Figure 16a, and  $m_2$ , shown in Figure 16b.

In Figure 16b, as a background, we show a series of colored vertical slices, whose widths correspond to the segments of  $m_2$ .

In Figure 16c, we transform the vertical slices determined by  $m_2$  into horizontal slices. On top of these horizontal slices, we show the map  $m_1$  again.

```

1  Fixpoint compose_2_aux m1 segs2 :=
2    match segs2 with
3    | SId gap =>
4      north gap m1
5    | Seg gap len segs2 =>
6      let x := invert m1 len in
7      clip x (north gap m1) (
8        let m1 := west x m1 in
9        compose_2_aux (north (-len) m1) segs2
10     )
11  end.
12
13 Definition compose_2 m1 m2 :=
14   compose_2_aux m1 (to_seglist m2).

```

Fig. 17. The second composition algorithm

In Figure 16d, we use *invert* to determine where these horizontal slices meet the graph of the function  $m_1$ . Using this information, we convert these slices into vertical slices.

The last step is to slide each slice of  $m_1$  towards the north by a suitable amount. To see this, and to find out by what amount each slice should slide, let us look again at Figure 16b. On the leftmost slice, the function  $m_2$  behaves like the diagonal function  $\lambda x. x + 0$ . On the second slice, it behaves like the diagonal function  $\lambda x. x + 4$ . On the third slice, it behaves like the diagonal function  $\lambda x. x + 7$ . On the last slice, it behaves like the diagonal function  $\lambda x. x + 12$ . Therefore, to obtain the composition of  $m_1$  and  $m_2$ , in Figure 16d, the leftmost slice should be moved up by 0 units; the second slice should be moved up by 4 units; the third slice should be moved up by 7 units; and the last slice should be moved up by 12 units. The result appears in Figures 16e and 16f.

**Code.** The code of this composition algorithm appears in Figure 17. This time, the map  $m_2$  is converted to a segment list, so that we can easily iterate over its segments.

For simplicity, we present a version of the code where *north* is used. By introducing an extra parameter  $gap_1$ , it is possible to eliminate the two calls to *north* that lie in the body of the loop, that is, in the second branch of the case analysis.

Let us examine the case where the segment list begins with a segment described by *gap* and *len*, followed with a sublist *segs2* (line 5). In the visual language of Figure 16b, this means that the leftmost vertical slice has width *len*. In terms of Figure 16c, this means that the bottommost horizontal slice has height *len*. We wish to determine where the upper limit of this slice meets the graph of the function  $m_1$ . To do so, we call *invert*  $m_1$  *len* (line 6). The result of this call, *x*, can be visually interpreted as the width of the leftmost vertical slice in Figure 16d. Using *north* and *clip*, we slide this vertical slice up by *gap* units and include it in the result of the composition. At this point, there remains to continue, via a recursive call to *compose-2-aux*. In this call, we move the map  $m_1$  towards the west by *x* units and towards the south by *len* units: this amounts to removing both the bottommost horizontal slice in Figure 16c and the leftmost vertical slice in Figure 16d from consideration.

```

1  Inductive view_ :=
2    | VId gap
3    | VSeg gap len (m : map).
4
5  Definition view (m : map) : view_ :=
6    match m with
7    | Id gap =>
8      VId gap
9    | Cons _ wght (Leaf len gap) jump m =>
10     VSeg gap len (north jump m)
11    | Cons wghm wght (Node _ gap len t1 t2) jump m =>
12     let wght := wght / 2 in
13     let m :=
14       Cons (wghm - 1) wght (tree_north gap t1) 0 (
15         Cons (wghm - 1 - wght) wght (tree_north gap t2) jump m)
16     in
17     VSeg gap len m
18  end.

```

Fig. 18. The function *view*

**Complexity.** The complexity analysis of this algorithm is largely analogous to our earlier analysis of the first composition algorithm (§4.6.1), where the roles of the two maps are exchanged. This time, the total cost of the calls to *west* and *clip* is bounded by

$$O \left( \begin{array}{l} \min(W_1 + W_2, W_2 \log W_1) \\ + \min(W', E_1, E_2) \end{array} \right)$$

The presence of a series of calls to *invert* is new, but their total cost can be bounded in the same way as the total cost of the calls to *west*. Therefore, the bound shown above is the worst-case time complexity of this composition algorithm.

This bound can be summarized as follows:

- (1) In all situations, the algorithm has complexity  $O(W_1 + W_2)$ .
- (2) If  $E_2$  is  $O(1)$  then the algorithm has complexity  $O(\log(W_1))$ .

#### 4.6.3 A hybrid composition algorithm

We have described two composition algorithms. The first algorithm (§4.6.1) cuts at most  $W_1$  vertical slices out of the map  $m_2$  and glues these slices together. It is especially efficient when  $m_1$  has small extent and therefore small weight. The second algorithm (§4.6.2) cuts at most  $W_2$  vertical slices out of the map  $m_1$ , slides them vertically, and glues them together. It is especially efficient when  $m_2$  has small extent and therefore small weight.

Can one get the best of both algorithms? A simple idea comes to mind: one should compare the weights of the two maps and, based on the outcome of this comparison, choose one of the two composition methods. As a refinement of this idea, this comparison and decision step can take place not just once, when the composition algorithm is invoked from the outside, but also every time the composition algorithm recursively invokes itself.

Our data structure supports *weight* queries with time complexity  $O(1)$ , so the weight comparison required by this hybrid algorithm costs nothing.

```

1  Fixpoint compose3_aux m1 m2 fuel :=
2    match fuel with 0 => Id 0 | S fuel =>
3    if fast_weight m1 <=? fast_weight m2 then
4      match view m1 with
5      | VId gap =>
6        west gap m2
7      | VSeg gap len m1 =>
8        let m2 := west gap m2 in
9        clip len m2 (
10         compose3_aux (north (-gap) m1) m2 fuel
11       )
12    end
13  else
14    match view m2 with
15    | VId gap =>
16      north gap m1
17    | VSeg gap len m2 =>
18      let x := invert m1 len in
19      clip x (north gap m1) (
20        let m1 := west x m1 in
21        compose3_aux (north_nz (-len) m1) m2 fuel
22      )
23    end
24  end.
25 Definition compose3 m1 m2 :=
26   compose3_aux m1 m2 (Z.to_nat (1 + weight m1 'min' weight m2)).

```

Fig. 19. The hybrid composition algorithm

**Code.** For simplicity, in our presentation of the first composition algorithm (§4.6.1), the map  $m_1$  was converted up front to a segment list. Similarly, in our presentation of the second algorithm (§4.6.2), the map  $m_2$  was converted to a segment list. In the hybrid algorithm, these conversions must be avoided, because they are too costly. Instead, we wish to decide on the fly whether to inspect the first segment of  $m_1$  or the first segment of  $m_2$ .

For this purpose, we define a function *view*, which provides a list-like view of a map. It appears in Figure 18. It is a *pop* operation on the random-access list. The time complexity of the function call *view m* is  $O(1)$ . This function call returns either *VId gap*, which means that the map contains zero segments (it is a half-line that begins at altitude *gap*), or *VSeg gap len m'*, which means that the map *m* is the horizontal concatenation of one segment (described by *gap* and *len*) with the submap *m'*.

Equipped with the function *view*, we may define the hybrid composition algorithm, whose code appears in Figure 19. Based on a comparison of the weights of the maps  $m_1$  and  $m_2$  (line 3), we use either the first or the second composition method. The function *fast-weight* reads a recorded weight and has time complexity  $O(1)$ . We distinguish it from the mathematical function *weight*, whose definition is recursive and whose time complexity would be  $O(W)$  if it was used at runtime. In each branch, a call to *view* lets us inspect just the first segment of the desired map.

To convince Rocq that the recursive function *compose-3-aux* terminates, we equip it with a parameter *fuel* of type  $\mathbb{N}$ . We prove *a posteriori* that  $1 + \min(W_1, W_2)$  is a sufficient amount of fuel, which means that the number of recursive calls is at most  $\min(W_1, W_2)$ . The quantity  $\min(W_1, W_2)$  decreases at least by one at each recursive call because *view* removes one segment from the smaller map. With luck, this quantity can decrease by more than one, as *west* (line 8 or line 20) removes zero, one, or several segments from the larger map.

Using a *fuel* parameter removes the obligation to prove termination at function definition time. An alternative approach would be to use well-founded recursion. That would force us to establish termination at function definition time, which would be significantly more painful than the present approach. Indeed, proving that the depth of the recursive calls is bounded by  $\min(W_1, W_2)$  requires exploiting the outcome of the weight comparison, which in turn requires arguing that the functions *fast-weight* and *weight* agree, which in turn requires keeping track of the fact that the maps  $m_1$  and  $m_2$  are well-formed. Proving that the depth of the recursive calls is bounded by  $W_1 + W_2$  (a weaker bound) would be easier but would still require polluting the code with dependent **match** constructs at lines 4 and 14.

For simplicity, the code in Figure 19 involves several calls to *north*, whose cost is  $O(1)$ . In order to save a constant factor in time and avoid several memory allocations, one can reformulate this code in such a way that it uses *north* only in the terminal cases where one of the maps has weight zero. We have written and verified this alternative formulation. Unfortunately, it is rather less readable, so we do not present it in the paper.

**Complexity.** The hybrid algorithm involves two sequences of cuts: one sequence out of the map  $m_1$  and one sequence out of the map  $m_2$ . This does not fundamentally affect its analysis. Its time complexity is bounded by

$$O \left( \begin{array}{l} \min(W_1 + W_2, \min(W_1, W_2) \log(\max(W_1, W_2))) \\ + \min(W', E_1, E_2) \end{array} \right)$$

In the first term, which represents the cost of cutting, we get the minimum of the previous two bounds: roughly speaking, the algorithm performs at most  $\min(W_1, W_2)$  lookups and reverse lookups in a map whose weight is at most  $\max(W_1, W_2)$ .

This bound can be summarized as follows:

- (1) In all situations, the algorithm has complexity  $O(W_1 + W_2)$ .
- (2) If  $E_1$  is  $O(1)$  then the algorithm has complexity  $O(\log(W_2))$ .
- (3) If  $E_2$  is  $O(1)$  then the algorithm has complexity  $O(\log(W_1))$ .

Because the identity map *Id* 0 has weight zero and extent zero, this complexity bound predicts that if  $m_1$  or  $m_2$  is the identity map then the composition algorithm terminates in constant time. Indeed this is true: in this case, because the weight of the identity map is zero, the algorithm applies *view* to the identity map, obtains *Id* 0 as a result, and terminates with a call to *west* 0 or *north* 0 (line 6 or 16), which itself returns immediately.

#### 4.7 Equality and identity tests

Although segment maps cannot be compared using a purely structural equality test (§3.5), implementing an equality test (*equal*) is straightforward. Our implementation of *equal* first tests, in constant time, whether the two maps have the same weight. If this is not the case, then it immediately returns *false*: indeed, two functions that do not have the same number of

discontinuities cannot be equal. If the two maps have the same weight, then, because the skew binary representation of a number is unique (Myers 1983), they must have the same shape. Then there remains to perform a comparison of the two maps, which can be described as a structural equality test, where the suspended northern displacements carried by every *Cons* constructor are applied on the fly. The time complexity of this comparison is  $O(W)$ , where  $W$  is the common weight of the two maps. As a special case, because *identity* has weight zero, the test *equal identity m* always returns in constant time.

The operation *is-identity-below x m* determines whether, on the interval  $[0, x)$ , the map  $m$  coincides with the identity function. The time complexity of this test is  $O(1)$ : indeed, because we require adjacent segments to be fused whenever possible, this query can be answered by inspecting just the first segment of the map  $m$ .

#### 4.8 Prune

*prune* expects an index  $x \in \mathbb{Z}^+$  and a map  $m$ . It returns a map  $m'$  such that the maps  $m$  and  $m'$  agree on the interval  $[0, x)$  and may disagree beyond  $x$ . The point is to discard unneeded information: presumably, because information can be discarded, the weight of  $m'$  can be less than the weight of  $m$ . In fact, *prune* guarantees that the weight of  $m'$  is  $O(W')$  where  $W'$  is the number of segments of the map  $m$  that overlap the interval  $[0, x)$ . This specification is non-deterministic: beyond index  $x$ , *prune* is free to retain or discard information.

An exact implementation of *prune*, which discards *everything* beyond index  $x$ , can be defined using *clip* and *identity*. However, it is expensive: its time complexity is  $O(W')$ . The weight of the map  $m'$  that is obtained in this way is exactly  $W'$ .

Fortunately, there is a cheaper, imprecise implementation. The idea is to walk the spine of the map  $m$  until the total width of the trees that are encountered reaches or exceeds  $x$ . There, the spine is cut: the trees that appear beyond this point are discarded. The time complexity of this implementation is  $O(\log W')$ , and the weight of the map  $m'$  that is obtained in this way is  $O(W')$ , as desired.

### 5 Discussion

We have presented segment maps, a representation of increasing functions of  $\mathbb{Z}^+$  to  $\mathbb{Z}^+$ , which supports a wide range of operations with good worst-case asymptotic time complexity. The paper is accompanied with OCaml and Rocq implementations of this data structure. The functional correctness of the Rocq implementation is machine-checked.

#### 5.1 Sources of inspiration and related data structures

In the design of this data structure, our starting point was the inductive definition of thinnings (McBride 2018). We compressed this representation by working with diagonal segments and vertical gaps of arbitrary (nonzero) length, as opposed to individual points. This led us to the concept of a segment list, which we further improved by replacing the linked-list structure with the structure of a random-access list (Myers 1983; Okasaki 1999).

We took inspiration from the representation of *explicit substitutions* that exists in the kernel of Rocq (The Rocq development team 2026). This data structure, attributed to Pierre-Marie Pédro, is described in a blog post by Bouverot-Dupuis (2026). It is based on Okasaki's random-access lists. It represents a substitution (a function of indices to terms), as opposed to a renaming (a function of indices to indices) or a thinning (an increasing function of indices

to indices). Therefore it is more general, but it does not have compression based on segments or an efficient composition operation.

In the same file, Rocq’s kernel defines a representation of thinnings, under the name of *explicit relocations* (The Rocq development team 2026). This representation looks superficially similar to a segment list, but is not a segment list. It is an inductive data type with three constructors, which in our terminology correspond to *identity*, *west*, and *northeast*. This is rather strange: indeed, intuitively, *west* discards information. If *west* is viewed as a *constructor* then inevitably the data structure is suboptimal, in the sense that it can store information that is no longer observable, and it does not enjoy canonical equality. To see this, suppose that one implements *north* via an application of the constructor *northeast* followed with an application of the constructor *west*. As a result, two successive calls to *north* 1 produce the sequence of constructors *west* 1/*northeast* 1/*west* 1/*northeast* 1. Yet these calls are logically equivalent to one call to *north* 2, which produces the sequence of constructors *west* 2/*northeast* 2. Therefore structural equality does not coincide with extensional equality.

In the same file still, Rocq’s kernel defines a representation of thinnings as segment lists, under the name of *weakenings*, and offers two functions that convert between “explicit relocations” and “weakenings”. This code is used for printing and debugging purposes only. It is due to Yann Leray (2025), who writes in a comment that segment lists are “more intuitive” than “explicit relocations”. We certainly agree! In the future, we would like to convince Rocq’s developers to switch to segment lists or (better) to segment maps. Furthermore, we would like to better understand how thinnings are used in Rocq’s kernel and to compare the performance of segment lists, segment maps, and bit vectors in this setting.

Allais (2023) mentions two operations on thinnings that we have not considered, namely *meet* and *join*, which, in the bit vector view, are bitwise “and” and bitwise “or” operations. In our view of a thinning as an increasing function of  $\mathbb{Z}^+$  to  $\mathbb{Z}^+$ , they can be understood as pointwise “min” and pointwise “max” operations on the derivative of the inverse map. It seems clear that segment lists support these operations with time complexity  $O(W_1 + W_2)$ , and as a consequence, so do segment maps. Quite plausibly, there exist more efficient implementations of *meet* and *join* on segment maps, which rely on cutting and gluing slices.

## 5.2 Real-world application and performance evaluation

The main questions that this paper leaves unanswered is: in practice, are segment maps really efficient? Are they superior to other representations of thinnings? We believe that investigating these questions in depth will require a substantial amount of work, which falls outside of the scope of this paper. The choice between segment maps and bit vectors (Allais 2023), which seem to be the main two contenders for an efficient representation of thinnings, is likely not clear-cut. Indeed, it depends on the manner and the scale at which thinnings are used. What are the typical weight and extent of the thinnings that are constructed in practice? What operations on thinnings are needed? If composition is used, what are the typical weight and extent of its two arguments? The answers to these questions depend on the intended application, which could be the implementation of a dependent type system, such as Rocq, Lean, Idris, Agda, F\*, smalltt (Kovács 2023), or a library that aims to offer an efficient implementation of terms with a high-level API, such as Rebound (Santo and Weirich 2025). The answers also depend on which representation of variables is chosen, such as de Bruijn indices (de Bruijn 1972) versus co-de Bruijn indices (McBride 2018).

### AI statement

Artificial intelligence was used neither in the work described in the paper nor in the writing of the paper.

### References

- Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. 1991. [Explicit Substitutions](#). *Journal of Functional Programming* 1, 4 (1991), 375–416.
- Andreas Abel, Guillaume Allais, Aliya Hameer, Brigitte Pientka, Alberto Momigliano, Steven Schäfer, and Kathrin Stark. 2019. [POPLMark reloaded: Mechanizing proofs by logical relations](#). *Journal of Functional Programming* 29 (2019).
- Guillaume Allais. 2023. [Builtin Types Viewed as Inductive Families](#). In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 13990)*. Springer, 113–139.
- Guillaume Allais, Robert Atkey, James Chapman, Conor McBride, and James McKinna. 2021. [A type- and scope-safe universe of syntaxes with binding: their semantics and proofs](#). *Journal of Functional Programming* 31 (2021).
- Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. 1995. [Categorical Reconstruction of a Reduction Free Normalization Proof](#). In *Category Theory and Computer Science (CTCS) (Lecture Notes in Computer Science, Vol. 953)*. Springer, 182–199.
- Andrew W. Appel and Trevor Jim. 1997. [Shrinking lambda expressions in linear time](#). *Journal of Functional Programming* 7, 5 (1997), 515–540.
- Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. 2008. [Engineering Formal Metatheory](#). In *Principles of Programming Languages (POPL)*. 3–15.
- Brian Aydemir and Stephanie Weirich. 2010. [LNgen: Tool Support for Locally Nameless Representations](#). Technical Report MS-CIS-10-24. University of Pennsylvania Department of Computer and Information Science.
- Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. 2005. [Mechanized Metatheory for the Masses: The POPLMARK Challenge](#). In *Theorem Proving in Higher Order Logics (TPHOLs) (Lecture Notes in Computer Science, Vol. 3603)*. Springer, 50–65.
- Jean-Philippe Bernardy and Nicolas Pouillard. 2013. [Names for free: polymorphic views of names and binders](#). In *Haskell symposium*. 13–24.
- Richard Bird and Ross Paterson. 1999. [de Bruijn Notation as a Nested Datatype](#). *Journal of Functional Programming* 9, 1 (Jan. 1999), 77–91.
- Mathis Bouverot-Dupuis. 2026. [Three implementations of explicit substitutions](#). Blog post.
- Mathis Bouverot-Dupuis, Théo Winterhalter, Kathrin Stark, and Kenji Maillard. 2026. [Sulfur: a Reflective Tactic for Substitution Simplification](#). In *Rocq for Programming Languages*.
- Arthur Charguéraud. 2012. [The Locally Nameless Representation](#). *Journal of Automated Reasoning* 49, 3 (2012), 363–408.
- James Cheney. 2005. [Scrap your nameplate](#). In *International Conference on Functional Programming (ICFP)*. 180–191.
- Nicolaas G. de Bruijn. 1972. [Lambda-Calculus Notation with Nameless Dummies: a Tool for Automatic Formula Manipulation with Application to the Church-Rosser Theorem](#). *Indag. Math.* 34, 5 (1972), 381–392.
- Lawrence Dunn, Val Tannen, and Steve Zdancewic. 2023. [Tealeaves: Structured Monads for Generic First-Order Abstract Syntax Infrastructure](#). In *Interactive Theorem Proving (ITP) (Leibniz International Proceedings in Informatics, Vol. 268)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 14:1–14:20.
- April Gonçalves and Wen Kokke. 2025. [Thinning Thinnings: Safe and Efficient Binders](#). In *Types for Proofs and Programs*.
- Andrew D. Gordon. 1993. [A Mechanisation of Name-Carrying Syntax up to Alpha-Conversion](#). In *Higher Order Logic Theorem Proving and its Applications (HUG 1993) (Lecture Notes in Computer Science, Vol. 780)*. Springer, 413–425.
- Andrew Kennedy. 2007. [Compiling with continuations, continued](#). In *International Conference on Functional Programming (ICFP)*. 177–190.
- Delia Kesner. 2009. [A Theory of Explicit Substitutions with Safe and Full Composition](#). *Logical Methods in Computer Science* 5, 3 (2009).

- Steven Keuchel, Stephanie Weirich, and Tom Schrijvers. 2016. [Needle & Knot: Binder Boilerplate Tied Up](#). In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 9632)*. Springer, 419–445.
- András Kovács. 2023. [smalltt](#).
- Dougal Maclaurin, Alexey Radul, and Adam Paszke. 2023. [The Foil: Capture-Avoiding Substitution With No Sharp Edges](#). In *Implementation of Functional Languages (IFL)*.
- Conor McBride. 2018. [Everybody’s Got To Be Somewhere](#). *Electronic Proceedings in Theoretical Computer Science* 275 (July 2018), 53–69.
- Eugene W. Myers. 1983. [An Applicative Random-Access Stack](#). *Inform. Process. Lett.* 17, 5 (1983), 241–248.
- Gopalan Nadathur and Xiaochu Qi. 2003. [Explicit Substitutions in the Reduction of Lambda Terms](#). In *Principles and Practice of Declarative Programming (PPDP)*, 195–206.
- Chris Okasaki. 1999. [Purely Functional Data Structures](#). Cambridge University Press.
- Simon Peyton Jones and Simon Marlow. 2002. [Secrets of the Glasgow Haskell Compiler inliner](#). *Journal of Functional Programming* 12, 4&5 (2002), 393–434.
- Emmanuel Polonowski. 2013. [Automatically Generated Infrastructure for de Bruijn Syntaxes](#). In *Interactive Theorem Proving (ITP)*. Springer, 402–417.
- François Pottier. 2021. [Strong Automated Testing of OCaml Libraries](#). In *Journées Françaises des Langages Applicatifs (JFLA)*.
- Noé De Santo and Stephanie Weirich. 2025. [Rebound: Efficient, Expressive, and Well-Scoped Binding](#). In *Haskell symposium*. 38–52.
- The Rocq development team. 2026. [rocq/kernel/esubst.ml](#).
- The Rocq team. 2026. [The Rocq Prover](#).
- Stephanie Weirich, Brent A. Yorgey, and Tim Sheard. 2011. [Binders unbound](#). In *International Conference on Functional Programming (ICFP)*, 333–345.