Une Table d'Association d'Intervalles Fusionnable

Richard Bonichon & Pascal Cuoq [1]

CEA LIST,
Laboratoire de Sûreté des Logiciels,
Boîte courrier 65,
Gif-sur-Yvette Cedex, F-91191 France
`{richard.bonichon, pascal.cuoq}@cea.fr`

Abstract. *This article describes an efficient persistent mergeable data structure for mapping intervals to values. We call this data structure* rangemap. *We provide an example of application where the need for such a data structure arises (abstract interpretation of programs with pointer casts). We detail different solutions we have considered and dismissed before reaching the solution of rangemaps. We show how they solve the initial problem and eventually describe their implementation.*

Keywords: *Rangemaps, interval maps, Patricia trees, abstract interpreation, sharing*

[1] Les travaux présentés dans cet article ont été réalisés dans le cadre du projet ANR U3CAT (convention ANR 2008-SEGI-021-1).

Résumé

Cet article explore le problème de la représentation de tables d'association persistantes indexées par des intervalles. Nous cherchons une structure de données qui permettent de trouver la valeur liée à un intervalle lorsque celui-ci a été inséré en tant que clé. Nous y ajoutons la contrainte additionnelle que cette représentation doit rester efficace dans cette recherche de valeur, même l'intervalle spécifique n'est pas une clé de la table d'association. Dans ce cas, on doit retourner toutes les valeurs associées à des intervalles intersectant l'intervalle demandé.

Un des exemples concrets d'application d'une telle structure de données concerne la modélisation des données d'un programme C durant son analyse par interprétation abstraite.

Dans ce cas, on ne peut se contenter de modéliser la mémoire comme un tableau de `unsigned char` dans le cas de valeur abstraites afin de pas perdre trop en précision. Soit un mot de 32 bits dont la valeur abstraite est $\{1, 258\}$. Les valeurs abstraites au niveau octet qui lui correspondent sont $\{1, 2\}$ pour l'octet de moindre poids, $\{0, 1\}$ pour son voisin et $\{0\}$ pour les deux derniers. Si on essaie de relire un mot à partir de cet endroit, les valeurs possibles sont maintenant $\{1, 2, 257, 258\}$! On ne peut donc stocker les octets des données indépendamment de leur relation dans le programme.

La structure de données doit rester générique et utiliser des fonctions de l'utilisateur pour recomposer les valeurs de façon adéquate. En particulier, tout boutisme (ou *endianness*) architectural, gros, petit ou inconnu, doit pouvoir être traité correctement.

Enfin, l'analyse doit pouvoir fusionner plusieurs branches dans lesquelles le même tableau de `unsigned char` a été modifié différemment. Afin de rester efficace, la structure de données doit être fusionnable au sens d'Okasaki et Gill [12], c'est-à-dire que les sous-structures doivent pouvoir être inspectées en utilisant une approche "diviser pour régner". Les résultats doivent pouvoir être mis en cache pour être utiliser plus tard sur des instances similaires de la structure de données. Notons également que la structure de données recherchée doit pouvoir supporter un partage maximal pour augmenter son efficacité.

À la recherche de la bonne structure de données

La solution que nous proposons dans cet article vient après avoir essayé et écarté d'autres représentations. Une des principales difficultés vient du fait qu'associer une valeur à un intervalle donné n'est pas la même chose que d'associer cette valeur à tous les éléments de cet intervalle. Il n'est donc pas possible d'utiliser une structure similaire aux *diets* [5].

La première idée qu'on peut avoir est d'utiliser des arbres AVL [1], comme ceux présents dans le module `Map` d'OCaml. Mais cette structure de données n'est pas fusionnable et les rééquilibrages interfèrent avec le partage maximal des données.

Les arbres Patricia [11] sont une solution élégante pour associer des valeurs à des entiers, lorsque la fusionnabilité est importante. À partir du moment où les clés peuvent être ordonnés suivant un ordre lexicographique, on peut construire un arbre Patricia.

On peut vouloir utiliser des arbres Patricia indexés par des intervalles. En représentant la table d'association comme un arbre Patricia gros-boutiste, on peut espérer que cela fonctionne, l'`id` de chaque intervalle étant par exemple sa borne inférieure. Mais les exemples pathologiques Fig. 1 et Fig. 2 montrent que lorsqu'on fusionne ces deux arbres, on ne peut “diviser pour régner”. Le calcul de la fusion ne peut être mémoïsé et cette représentation ne correspond donc pas au critère de fusionnabilité recherché.

Si on essaie maintenant d'indexer les arbres Patricia par des entiers, on obtient une alternative intéressante à la solution que nous proposons. En omettant les nœuds dont les fils pointent au final vers la même feuille, on obtient une structure approchant des diagrammes de décision binaire [2] travaillant sur les bits des clés. Les feuilles de cet arbre seraient liées à un préfixe de clé au lieu de la clé entière elle-même. Un écueil de cette approche est qu'elle ne fonctionne bien que lorsque la représentation binaire des entiers de l'intervalle-clé est caractérisée par un nombre faible de préfixes communs. L'intervalle $\{0..47\}$ correspond à deux liens (pour $\{0..32\}$ et $\{33..47\}$) mais $\{0..62\}$ en a six!

Notre solution

Les tables d'association que nous proposons, baptisées *rangemaps* associent des valeurs aux intervalles avec une fusionnabilité similaire aux arbres Patricia indexés par des entiers. Les associations y sont enregistrées sur les nœuds, contrairement aux arbres Patricia et comme dans les *diets*. D'ailleurs si les valeurs stockées sont des booléens, cette structure devient assez comparable aux *diets*, avec la propriété additionnelle d'être fusionnable.

Dans les rangemaps, les nœuds sont ordonnés statiquement comme dans les arbres Patricia. Celui dont l'intervalle contient un multiple d'une plus grande puissance de deux sera placé au-dessus de l'autre. Cet ordre permet également d'avoir un certain équilibrage automatique de l'arbre. On peut voir un arbre ordonné de cette façon en Fig. 4.

La fusionnabilité des rangemaps est supérieure à celle des arbres Patricia indexés par des intervalles. En fait, même si les premiers paraissent souffrir du même problème que ces derniers, chercher les associations adjacentes est limité aux nœuds parents immédiats. Ceci est en fait une conséquence l'ordre statique utilisé.

Orthogonalement, et dans le cadre de la recherche de partage maximal au sein d'un même arbre, les sous-arbres des rangemaps sont tous relatifs. Les flèches de l'arbre portent sur elles les décalages du fils par rapport au père (cf Fig. 5).

Afin d'optimiser la compacité de la représentation, les intervalles adjacents sont automatiquement recousus en un seul lorsque les valeurs auxquelles ils sont liées sont identiques. Cette optimisation est une idée empruntée aux *diets* d'Erwig [5] et étendue aux valeurs plus complexes que portent les rangemaps.

Ces optimisations se traduisent lors de l'implantation par des fonctions intelligentes de construction et d'insertion d'éléments qui respectent les invariants d'ordre et de compacité dont nous avons parlé. En particulier on peut voir comment fonctionne une insertion provoquant un recousement Fig. 6 et Fig. 7. Comme ces fonctions intelligentes, la recherche des valeurs liées à un intervalle, dont nous avons parlé en introduction au problème, est implantée en utilisant les zippers de Huet [9]. Ces zippers sont également utiliés lorsqu'on a besoin de naviguer à l'intérieur des rangemaps ou lorsqu'ils ont besoin d'être recousus.

Cette structure de données est actuellement mise à l'épreuve des tests de Frama-C dans le but final d'améliorer la précision et l'efficacité (en mémoire et vitesse) de l'analyse de valeur. Parce que cette re-implémentation est aussi l'occasion de rationaliser certains des algorithmes utilisés, les résultats préliminaires montrent un gain intéressant de précision. Par contre, la structure précédemment utilisée ayant accumulé une quantité de fonctionalités conséquente qui doivent toutes être reproduites avant de pouvoir la débrancher, il n'est pas encore possible à l'heure actuelle d'effectuer de comparaisons de performances avec la nouvelle représentation.

1. Introduction

The problem at hand

The problem we are concerned with in this article is the representation of persistent maps indexed by intervals. The solution we are looking for is a data structure that allows to retrieve the value associated to an interval when this interval has previously been used as the key in an insertion; an additional constraint is that the representation must also be efficient for finding what an interval is mapped to when this specific interval is not employed as a key in the map. In this case, the data structure must allow to retrieve all bindings that intersect the required interval, and let these partial results be combined to the programmer's liking.

As a concrete example, consider the modelization of the contents of an `unsigned char` array during the abstract execution of a C program. We are concerned with the analysis of embedded C code, for which target endianness and word size are known, portability is not a concern, and the code often has no choice but to contain low-level constructs (e.g. [10]). Type-punning is used in this introductory example. Some compilers assert the right to mistranslate such C code. Embedded compilers do not: compiling this kind of code is precisely what they are used for. The same example could have been written using unions. The result would then still be undefined, but most compilers offer guarantees in addition to the base undefinedness of the C99 standard in this case.

The C program may take the address of any cell in this array and cast this address to an `int*`. If the run-time architecture allows it, the program may then use this pointer to write an `int` (on a 32-bit architecture, the `int` occupies four consecutive cells in the original char array). If the same pointer (as cast to `int*`) is now dereferenced, it is desirable to recover exactly the same abstract value that was previously written there. If an `unsigned char*` pointer referencing one of the four aforementioned cells is dereferenced, the result should be that the value read is a part of the stored `int`. And lastly, if an 8-byte `double` is read from the same location, the data structure should be able to indicate that the bits read are made partly from the `int`, and partly from other values, each of which may for instance have been written as a `char` previously.

In a concrete (say, hardware) implementation, the memory can be considered as an unadorned byte array. A multi-byte memory access reads – or writes – several consecutive bytes, and that's all there is to it. In this case, there is no need to consider the memory as a map whose keys are intervals. But in the case of abstract values, an unfortunate loss of precision would occur if the same approach was employed. Consider as an example of abstract value for a 32-bit word the pair $\{1, 258\}$. Projecting this word-level abstract value in abstract values for the component bytes, we obtain the set $\{1, 2\}$ for the least significant byte, $\{0, 1\}$ for its neighbor, and $\{0\}$ for the two most significant bytes. Now trying to read back a word from the same location in memory, it appears that the possible values for the data word are $\{1, 2, 257, 258\}$, an unacceptable approximation. Obviously, the component bytes of a data word can not be stored independently without concern for the implicit relationship between them.

For maximum generality, the data structure should not try to decide how to read a single `unsigned char` from a stored `int`, or how to recompose a `double` from several smaller values. Instead, it should be generic and call user-provided functions to recompose values when appropriate. Specifically, the structure should be generic enough that, by providing the right functions, little, big, and unknown endiannesses can all be accommodated.

Finally, C being an imperative programming language, the analysis may involve merging together several execution branches where the same `unsigned char` array has been modified in different ways. For good performance, the data structure should be mergeable, in the exact same sense as in Okasaki and Gill's [12]. We informally define mergeability as the following property:

Property 1 (Mergeability)**.** *When iterating in parallel on two instances of the data structure, a divide-and-conquer approach allows to consider separately the component substructures. Additionally, the results obtained for these substructures can be cached, and have a good chance to be useful later, when processing other instances that are only slightly different from the initial ones.*

The mergeability property is named in reference to the `merge` operation that takes two trees t_1 and t_2, and builds a tree where each key contains the merge of the values associated to this key in t_1 and t_2. The `merge` function can quickly check the subtrees that appear at the same level in t_1 and t_2 for physical identity. If these subtrees are physically equal, the computation of their merge is immediate (the result subtree is in this case identical to the arguments). This can happen for instance when t_2 was created by slightly modifying t_1, or, if hash-consing [3, 4] is used to ensure maximal sharing, at every opportunity.

Contents of the article

This article shows the development of an efficient solution to the problem we have exposed. A starting point for the reflection is the Patricia tree structure, summarized in Sect. 2. We show in Sect. 3 how we considered various other solutions based upon AVL trees (Sect. 3.1) and Patricia trees (Sect. 3.2 and Sect. 3.3). This eventually leads us to describe the data structure answering our needs in Sect. 4. We have a prototype implementation of rangemaps, from which we have extracted relevant technical details in Sect. 5. Considerations about the future of this work conclude this article in Sect. 6.

Note that most of the article is language agnostic but code samples are in OCaml.

2. Reminder: Patricia trees

Morrison's Patricia trees [11] provide a great technical solution for mapping integers to values, when mergeability is important. In fact, Patricia trees are rather unique in this respect, and they will serve as a natural guide in our reflection.

Patricia trees are used to implement maps when there exists a natural lexicographical order on the keys. When used with integers as keys, the lexicographical order used is the comparison of the keys' binary representations. Okasaki and Gill[12] offer both "little-endian" and "big-endian" versions — these terms characterizing here the order in which bits in a key are scanned, either from most significant to less significant or the other way round. If the "big-endian" representation is chosen, and assuming we have to deal only with positive integers (both assumptions we will make in the remainder of this article), the lexicographical order on binary representations coincides with the usual order of integers.

During the lookup of a key k in a Patricia tree, each node, starting from the root, tests one bit in the binary representation of k in the lexicographical order. When, for a given prefix, all keys present in the tree have the same value for the next bit, the comparison of this bit is skipped, so that in general, it takes about $log_2(n)$ comparisons to get the value associated to a key in a map of n bindings. There never is any rebalancing in Patricia trees: the nodes of the tree are hierarchized according to the lexicographical order that has been fixed in advance. Patricia trees can therefore be unbalanced (a worst-case example is a map where the keys are 1, 2, 4, 8, 16... This map is represented as a comb). However, the height of a Patricia tree is bounded by the base-2 logarithm of the difference between the values of its smallest and largest key. If arbitrary 32-bit integers are used as keys, a Patricia tree can never have an height higher than 32 (and a lookup never require more than 32 comparisons).

When maps indexed by a type `key` different from `int` are required, it is still sometimes possible to employ Patricia trees. It may be a simple matter of tagging each object of type `key` at creation with a unique integer (*e.g.* an `id` field in a record type). This makes it possible to use Patricia trees for representing maps from keys to values, by using the `id` field as the actual key during lookups.

However, these maps may be expected to provide primitives that give access to the original keys, e.g. a function `mapi: (key -> ’a -> ’b) -> ’a t -> ’b t`. These primitive can be made possible by storing actual keys (of type `key`) at the leaves of the tree, in place of the integer keys stored in traditional Patricia trees implementation, knowing that when the integer is needed, it can be obtained from the key.

3. Towards mergeable binary trees indexed by intervals

Consider now the problem of mapping non-overlapping intervals of integers to values. A concrete use case for this structure was provided in Sect. 1. The same property of mergeability that Patricia trees have is expected of this data structure.

Another use case is to implement sets of integers as maps from integers to booleans, with the same property that diets [5] have, of representing consecutive sequence of present (resp. absent) integers as a single binding. Diets do not have the mergeability property; if the reader is familiar with this data structure, he may prefer to think of the problem as that of representing mergable diets.

A data structure indexed by intervals

For the "abstract interpretation of low-level C code" example, mapping an interval to a value is not the same thing as mapping all integers in the interval to this value. For the "mergeable diets" example, the distinction is not necessary.

For maximum generality, the data structure should not make assumptions about what happens when a new binding added to a map overlaps with some of the bindings already in place. Behaviors that may be useful are to remove the overwritten binding completely, or to alter the contents of the untouched parts of the original binding to reflect the fact that they were part of a larger binding that was partially overwritten. Providing these behaviors implies for instance that the binding $0..31 \rightarrow \{22\}$ is not the same thing as 32 individual bindings $0 \rightarrow \{22\}$, $1 \rightarrow \{22\}$,...

The solution data structure should, for instance, be able to answer queries such as finding all bindings that intersect a given interval. Being able to do this is necessary for lookups, but also at the time of adding a new binding, in order to maintain the invariant that the intervals in the map are disjoint.

While it is possible to associate a unique integer to use as an `id` to any interval, this solution, applied naively, would store the intervals in the map without respect for their natural order, and as a consequence, the whole tree would have to be explored in order to find the bindings that intersect a given interval.

3.1. *Using AVL trees with the natural interval order*

Note that the intervals used as keys in a given map are guaranteed to be disjoint. The first idea is therefore to order them according to their natural order, which happens to be total among the keys of a single tree. Many data structures have been proposed for representing maps indexed by totally ordered keys as trees. In most of them, the tree structure can be taken advantage of to quickly look up a range of bindings. As an example, the `Map` module in the OCaml standard library (implemented with AVL trees [1]) can easily be augmented with functions for looking up all bindings that intersect the query interval. The resulting data structure, unfortunately, is no more mergeable than `Map` usually is. Balancing operations interfere with sharing.

3.2. *Trying to use Patricia trees indexed by intervals*

Since, for us, mergeability is an important criterion, and since Patricia trees have a reputation for being mergeable, it is natural to try to representing the map as a big-endian Patricia tree, using for instance the lower bound of each interval as its `id`. It may be possible to make this representation work, but mergeability again suffers unexpectedly. Consider indeed the example on Fig. 1.

In Fig. 1, the binding $0..47 \rightarrow \{11\}$ is stored in the leftmost binding, but it contains information that may interfere with a differently located binding in another tree.

When computing the merge of the two trees from Fig. 1 and Fig. 2, the divide-and-conquer approach does not work! The 0..47 binding in the left-hand-side subtree of Fig. 1 contains information that is relevant for computing other branches of the result. Therefore, the computation of the merge of such affected subtrees cannot be cached, since it depends on external factors. This choice of representation is simply not mergeable in the sense of Okasaki and Gill [12].

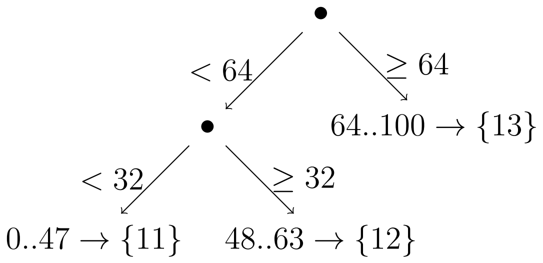

Figure 1 – Interval map as a Patricia tree, using the minimum bound as id

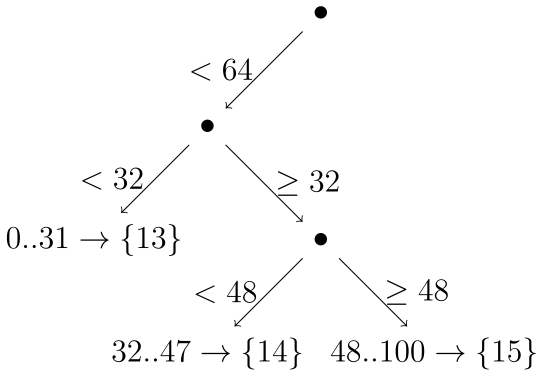

Figure 2 – Another interval map represented as a Patricia tree

3.3. *Trying to use Patricia trees indexed by integers*

Let us now proceed to show an alternative representation not quite like the one we are proposing, which could nevertheless be a reasonable alternative to it. Patricia trees can be made to omit the nodes whose children both eventually lead to the same leaf anyway (making them even closer than they already are to Binary Decision Diagrams [2] working on the bits of the key). We have not encountered this optimization in the literature, perhaps because it requires comparing the values that are bound to the keys, which may be expensive outside the context of hash-consing. This optimization can be done by changing the `Leaf` constructor to contain a prefix instead of a key (meaning that in this tree, all keys with this prefix are bound to the same value). Then, it is only a matter of systematically replacing applications of the `Node` constructor by a "smart constructor" function which checks whether the left and right subtrees are both `Leaf` with the same value, and replaces the Node and its two children with `Leaf` in this case.

With this optimization enabled, Patricia trees can efficiently represent identical bindings to long consecutive sequences of keys. It becomes feasible to represent the $0..47 \rightarrow \{12\}$ binding as a binding from each of the integers 0, 1,$\ldots$, 47 to any single value. This single value should of course contain the bound value $\{12\}$, but also the interval that serves as key of this binding, so that an access inside the interval (say, to the value bound to the interval $8..15$) is allowed to recover the information that this binding is part of a sequence that goes from 0 to 47. In particular, adding a new binding to the interval $8..15$ in this tree should either transform the existing $0..47 \rightarrow \ldots$ binding into three bindings $0..7$, $8..15$, $16..47$ or into a single $8..15$ binding depending on the desired semantics for overlapping writes. In both cases, it is necessary to have the information that the current binding at 8..15 is really a sub-part of a binding to a larger interval, so that this binding can be completely modified or removed. It would be necessary to make use of zippers[9] in order to navigate efficiently from the bindings at 8..15 to the adjoining 40 bindings that used to be related to them.

As a foreseen drawback with this approach, note that the factoring of identical bindings suggested here only works well when the binary representations of the integers contained in the key interval are characterized by a few common prefixes. A key such as $0..47$ would require two actual bindings in the tree to be represented (for $0..32$ and $33..47$). A worst-case interval such as $0..62$ woud require 6 actual bindings to represent (for $0..31$, $33..47$, $48..55$, $56..59$, $60..61$ and $62..62$).

4. The solution we propose

We propose to build maps from intervals to values, with the same mergeable quality that Patricia trees display for integer-indexed maps. As a first difference from Patricia trees, but similarly to diets [5], in our proposal bindings are recorded on the nodes, whereas Patricia trees record bindings at the leaves. Diets are a data structure to represent sets, when a total order, and successor and predecessor functions are available for the elements – for instance, integers. Diets are efficient when long sequences of consecutive elements commonly occur. On the other hand, our proposal is a data structure for maps indexed by intervals of integers, but using this structure to map intervals to a boolean gives an implementation for sets of integers which it is instructive to compare to diets.

4.1. *Basic idea*

Let us assume for simplicity that we are only interested in representing interval-indexed maps in which the keys, in addition to being non-overlapping, are contiguous and always cover the same definition interval. We will always use the interval 0..100 in the examples. For consistency with this invariant, a new map will contain a single binding from the key 0..100 to a value provided at construction time: `val new_map : ’a -> ’a tree`.

Such a map is represented in our data structure by a single node with empty subtrees (see Fig. 3). From this point onwards, for the sake of readability, when both subtrees of a node are empty, we omit them from the figure.

A function `add` allows to change part of an existing map:

```
val add : int*int → 'a → 'a tree → 'a tree
```

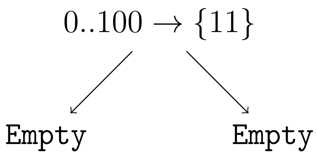

Figure 3 – Tree for the map $0..100 \to \{11\}$

Let us consider what happens when calling `add (20,30) {12}` on the initial tree created above. The resulting map has bindings $0..19 \to \{11\}$, $20..30 \to \{12\}$, and $31..100 \to \{11\}$. In fact, we must not confuse the bindings at $0..19$ and $31..100$ for bindings containing the value $\{11\}$: they are both remaining parts of a binding that originally spanned the interval $0..100$ and has been partially overwritten. In order to make this distinction explicit, we denote the map as $0..19 \to \{11\}_{0..100}$, $20..30 \to \{12\}$, and $31..100 \to \{11\}_{0..100}$

It is obvious how to arrange these bindings in a tree for easy retrieval: with the lower bindings on the left-hand-side and the higher bindings on the right-hand-side. What is not obvious is deciding which node goes on top so that the tree ends up balanced or nearly balanced.

One possibility is to record the height of the trees and to build trees that are balanced by construction. This amounts to using AVL trees [1] with the natural order on intervals, which we proposed as an ad-hoc solution in Sect. 3.1. Unfortunately, the re-balancing operations cause the creation of physically different trees that contain the same sets of bindings, that is, loss of mergeability.

Another possibility, which preserves mergeability, relies on the same idea that underlies Patricia trees. In Patricia trees, there is a static hierarchy for deciding which node goes above the other, and this static hierarchy ensures trees are balanced or almost balanced without any re-balancing operations. In the case of our data structure, we similarly define a static ordering on intervals that tells which node must be placed above the others. Intuitively, the interval containing the multiple of the largest power of two is put at the top. Like the ordering in Patricia trees, this ordering has the bounded chain length property (the bound is the

2-logarithm of the definition interval's width, give or take a couple of units).

This ordering uses a notion of *rank* for the intervals used as keys. Of two intervals for which we must decide which should be the parent of the other of the other, the one to go on top is the one with the highest rank. Let us now define this notion of rank more formally.

Definition 1 (Rank of an interval)**.** The rank of an interval I is defined as:

$$rank(I) = (k \mid \exists x \in I \; x \bmod 2^k = 0 \wedge$$
$$(\forall y \in I \forall k' \; y \bmod 2^{k'} = 0 \Rightarrow k' \leq k))$$

In particular, it follows from Def. 1 that any interval containing 0 has any rank, because $\forall k, \;\; 0 \bmod 2^k = 0$. We take as convention that the interval containing 0 will always have the highest rank of all intervals contained in a tree (this special value will be denoted as ∞). From the definition of a rank, we can now define a strict partial order on our intervals.

Definition 2 (Strict partial order over intervals, $\succ_i$)**.** Let I_1 and I_2 be intervals.

$$I_1 \succ_i I_2 \iff rank(I_1) > rank(I_2)$$

This strict partial order has the additional property that two contiguous intervals are always comparable.

Lemma 1 (Comparability of adjacent intervals)**.** *Let I_1 and I_2 be (non-equal) adjacent intervals. Either $I_1 \succ_i I_2$ or $I_2 \succ_i I_1$.*

Proof. Let $I_1 = a_1..b_1$ and $I_2 = a_2..b_2$ with $a_2 = b_1 + 1$.
Now assume $rank(I_1) = rank(I_2)$, *i.e.* $\exists n_1, n_2, k, \;\; n_1 2^k \in I_1 \wedge n_2 2^k \in I_2$ with $n_1 < n_2$.

We know that $\exists n \; n_1 \leq 2n \leq n_1 + 1 \leq n_2$ (one of two consecutive integers is even). Hence $\exists n, \;\; n_1 2^k \leq 2n 2^k = n 2^{k+1} \leq n_2 2^k$. Either I_1 or I_2 contains the value $n 2^{k+1}$ as they are contiguous, therefore either $rank(I_1)$ or $rank(I_2)$ is $k + 1$, contradicting our first assumption. □

Fig. 4 shows the tree representation ordered with the rank function for the following bindings: $0..19 \rightarrow \{11\}_{0..100}$, $20..30 \rightarrow \{12\}$, $31..100 \rightarrow \{11\}_{0..100}$.

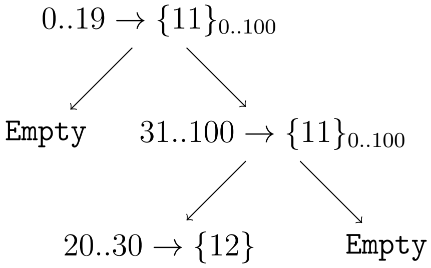
$0..19 \rightarrow \{11\}_{0..100}$
Empty
$31..100 \rightarrow \{11\}_{0..100}$
$20..30 \rightarrow \{12\}$
Empty

Figure 4 – Ordering nodes according to rank

In Fig. 4, the interval $0..19$ is at the root because it contains 0. The interval containing 0 is always at the root by convention. The interval $31..100$ contains $64 = 2^6$ whereas the interval $20..30$ contains $24 = 3 * 2^3$, therefore the former goes above the latter. The $20..30$ binding ends up as left child of the $31..100$ binding.

4.2. *Pretty well mergeable*

In Sect. 3.2, we claimed that Patricia trees in which lower bounds of intervals were used as ids did not fit the mergeability constraint because during a recursive descent on two separate trees, corresponding subtrees would have different definition domains. When trying to merge the trees in Fig. 1 and Fig. 2, for instance, one encounters the problem that the corresponding leftmost subtrees contains the bindings for $0..47$ in one tree and $0..31$ in the other. To merge these subtrees, in practice, it is necessary to borrow the contents of the $32..47$ range from the context of the second tree.

The attentive reader may have noticed that the solution we are proposing appears to suffer from a similar problem. During a recursive descent of separate rangemaps (for instance in the context of a merge operation), the definition domains for encountered subtrees may differ too. This is in fact unavoidable, as the partitioning of the definition domain into intervals may not match at all between the two trees. Also with rangemaps, it may be necessary to patch the narrower subtree to extend its definition domain to the same size as the other, in effect borrowing bindings from its context.

The important difference is that in rangemaps, with the interval ordering that we defined, the "context" in which it is necessary to look for bindings to borrow is limited. Specifically, *only the most immediate ancestor node from which we descended to the right, or the most immediate ancestor from which we descended to the left, to the exclusion of any other, need to be borrowed from*. Both on the left-hand-side and on the right-hand-side, there is at most one binding to move temporarily to the narrowest subtree to equalize it. Because of the way the static interval ordering works, it is never necessary to look further than this parent.

To illustrate this claim, consider the example of two corresponding rank 5 subtrees t_1 and t_2 (let us assume each subtree's root binding contains 32). The subtree that reaches the farthest to the right, t_1, can not span past 63. On the other hand, t_2's parent is the binding that contains 64, and therefore, it is not necessary to look elsewhere than in this parent node to get a piece of binding that equalizes the definition domain of t_2 with that of t_1.

By contrast, in the solution from Sect. 3.2, an arbitrary number of bindings may have to be borrowed to equalize the definition domains.

4.3. *Relative subtrees*

An orthogonal optimization, mentioned here for completeness, is to make all subtrees relative. The arrows between the nodes carry offsets that must be tracked when traversing the tree. The benefit obtained in exchange for this additional complexity is that sharing (see also Sect. 5.6) becomes possible within a single, repetitive map, in addition to the sharing between distinct but similar maps that other tree representations usually allow. Note that this optimization is not specific to rangemaps and can be adapted to most kinds of trees with numerical keys.

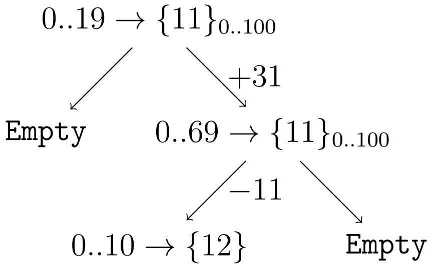

Figure 5 – The same map as in Fig. 4, represented with relative subtrees

4.4. *Automatic stitching of identical adjacent bindings*

Yet a different, complementary optimization for compact representation of repetitive trees is to automatically stitch adjacent bindings to the same value into a single, wider binding. This is in the spirit of what Erwig proposed for diets [5]. However, because our data structure is a map indexed by intervals, requirements for stitching are more sophisticated. Values must be identical, but also be stored with the same width, and the right-hand side value must start exactly where the left-hand side one ends.

To make this optimization easily applicable, we store the information about the original span of the binding (that we denoted as a subscript $\{11\}_{0..100}$ in previous examples) in the form of a binding width (for this example, 101) and an offset (for this example, 0).

An example of binding that can be stitched to this one is $101..201 \to \{11\}$. It starts where the $0..100$ binding finishes, and it contains the same value (with the same width). With the (width, offset) representation, the criterion for recognizing that two bindings are stitchable is that values, widths, and offsets are identical for both, and that in addition, the stitching point (here 101) is congruent to offset modulo width (that is, the point of stitching is actually a point where a value ends and a new one can start). Stitching occurs in particular when adding a new node to an already existing rangemap: this is illustrated in Fig. 7.

In the use case of abstract interpretation of C programs, it is clear why it is undesirable to omit the last condition above: on a little-endian architecture, a binding $0..0 \rightarrow \{1, 258\}_{0..3}$ may be the abstract result of taking the first byte of the concrete 32-bit value 1. The binding $1..3 \rightarrow \{1, 258\}_{0..3}$ may be the abstract result of taking the last three bytes of the concrete 32-bit value 258. Stitching these two bindings together into a single binding $0..3 \rightarrow \{1, 258\}$ would be incorrect: the value contained in these four bytes in a concrete execution, 257, would not be represented by the abstract value $\{1, 258\}$ resulting from the stitching.

On the other hand, when the stitching occurs at a point where a binding ends and another starts, no incorrectness results from stitching them together. Indeed, these bindings are still considered as different bindings after the stitching has occurred.

In effect, the optimization proposed in this section consists in enforcing the invariant that "no two adjacent stitchable bindings coexist in the rangemap". Therefore, whenever a binding is added, or changed, in a rangemap, the adjoining bindings must be checked for stitching possibilities.

5. Implementation notes

We have implemented proof-of-concept rangemaps, and we hope to soon be able to substitute with this implementation the existing, ad-hoc interval-indexed maps in the value analysis of Frama-C [6]. This section describes the OCaml implementation. Most functions in the implementation of rangemaps follow a divide-and-conquer pattern. Therefore, they can be cached, in the hope that partial results from previous similar computations can be reused. In the context of Frama-C's value analysis, rangemaps can be expected to exhibit sharing both because of the existence of maps that are slight variations of each other, and because of repetitive bindings within a single map. Caching allows to take advantage of spatial sharing to gain in execution time.

5.1. *Datatype*

Rangemaps are trees built from the following algebraic datatype:

```
type t =
    | Empty
    | Node of Int.t *
      (* max:  min is always implicitly zero *)
              Int.t * t *
      (* offset_left * subtree_left *)
              Int.t * t *
      (* offset_right * subtree_right *)
              Int.t * Int.t * V.t
      (* offset * modulo * value *)
```

Tree nodes carry the following information:

– the length of the interval (max+1);

– where (`offset_left`, `offset_right`) and what (`subtree_left`, `subtree_right`) its left and right children are. These offsets are computed as the difference between the lower bound of the child and that of the parent;

– the data bound to the interval, i.e a value repeated each `modulo` starting from `offset`.

5.2. *Zippers and tree browsing*

The implementation of rangemaps needs to allow for easy browsing through the nodes of the tree. Huet's zippers [9] are used in the implementation precisely to efficiently go from one node to its neighbors. They are also used during the construction of values in Sect. 5.3.

```
(** Zippers: Offset of a node *
             Node *
             continuation of the zipper *)
type zipper =
 | End
 | Right of Int.t * t * zipper
 | Left of Int.t * t * zipper;;

exception End_reached;;
```

The End_reached exception is used to signal we have reached the end of a zipper during a rezipping phase.

Actually, we only need to be able to go from one interval to the next adjacent one in increasing order. This function, called `move_right`, is central to many operations on the tree.

There are only two cases:

– either there is a node below on the right-hand child of the current node and we only have to grab the leftmost child of this child (which contains the next adjacent node);

– or there is none and this means the next adjacent node is above the current node; we therefore need to go back using the history of navigation contained in the zipper until we see a left turn, which means we were previously looking for a lesser node than the one where we turned left; in such a case, the node where the zipper indicates a left turn is the right one.

The implementation in OCaml is straightforward and goes as follows:

```
(** Move to the right of the current node
    Uses a zipper for that
 *)
let rec move_right curr_off node zipper =
 match node with
 | Node (_, _, _, offr , ((Node _ ) as subr), _, _, _) →
    let new_offset = add curr_off offr in
    (* if there is a right child
       the next adjacent one
       is the leftmost child of this right child *)
   leftmost_child
       new_offset
       (Right (curr_off , node , zipper))
       subr
 | Node (_, _, _, _, Empty , _, _, _) →
 (* otherwise we should go up until
    we see we have taken a left turn
    whenever that happens we have found
    the next adjacent node
 *)
   begin
    let rec unzip_until_left zipper =
     match zipper with
     | End → raise End_reached
     | Right (_, _, z) → unzip_until_left z
     | Left (offset , tree , z) → offset , tree , z
     in unzip_until_left zipper
   end
 | Empty → assert false
  ;;
```

The arguments needed by `move_right` are the current offset of the considered node and a zipper which corresponds to the path taken up to this node. The function returns a tree with its offset, as well as the corresponding zipper to get there.

5.3. *Construction of values*

Construction of values of the type `t` is done exclusively using so-called "smart constructors". This enforces various invariants including that of Sect. 4.4. Only the module implementing the rangemaps has direct access to the algebraic constructors (`t` is abstract but could as well have been declared `private`).

The stitching phase of Sect. 4.4 makes use of the zippers of Sect. 5.2 : they represent the context in which the subtrees of the node to be stitched should be re-attached (see also Fig. 7).

We will concentrate here on the `add_binding` function of the module, which internally calls a smart `make_node`. Their signatures, as implemented, are:

```
val add_binding :
        int64 → int64 → int64 → Int.t → Int.t →
        V.t → t → int64 * t
(* [current_tree_offset] →
   [min] → [max] → [off] → [modu] → [value] →
   [tree] → [new_current_tree_offset] * [current_tree]
*)

val make_node :
        int64 → Int.t → Int.t → t → Int.t → t →
        Int.t → Int.t → V.t → int64 * t
(* [current_tree_offset] → [max]  →
   [offset_left_subtree] → [left_subtree] →
   [offset_right_subtree] → [right_subtree] →
   [off] → [modu] → [value] →
   [current_new_tree_offset] * [new_tree]
*)
```

Let us illustrate how the smart `add_binding` operates on the rangemap shown in Fig. 6, which represents the following sequential intervals: $0..19 \rightarrow \{11\}$, $20..30 \rightarrow \{12\}$, $31..65 \rightarrow \{11\}$, $66..80 \rightarrow \{14\}$, $81..88 \rightarrow \{15\}$, $89..100 \rightarrow \{13\}$ with respective ranks ∞ (by convention), $3\ (24 = 3 * 2^3)$, $6\ (64 = 2^6)$, $5\ (80 = 5 * 2^4)$, $3\ (88 = 11 * 2^3)$, $5\ (96 = 3 * 2^5)$. As an added hypothesis, we suppose all

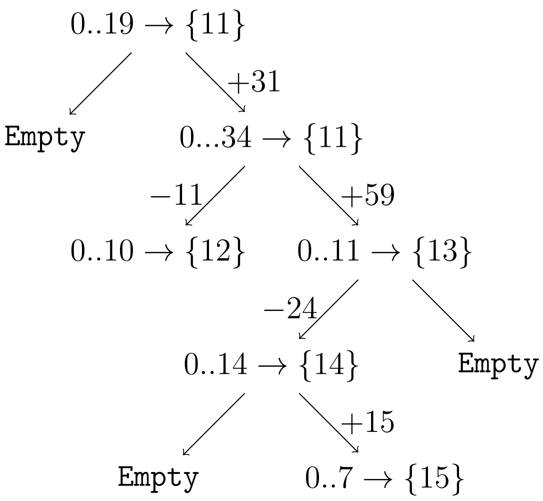

Figure 6 – Initial tree before addition of a binding

off and modu are given in such a way that the adjacent intervals of this rangemap can be stitched together provided they hold the same value.

Suppose that we now want to add the following binding $31..65 \to \{14\}$ to this rangemap of Fig. 6. The operation can be decomposed as follows (see also Fig. 7):

1) Find the correct spot according to rank (Def. 1) where the new node should be;

2) See if it can be stitched together with some node of its subtrees;

3) Call make_node and rezip if needed, stitch if needed.

5.4. *Merging rangemaps*

From the get-go, we were aiming at a mergeable structure and our module of course contains a suitable function with the following immediate signature:

```
val merge: int64 → t → int64  → t → int64 * t
(* [offset_t1] → [t1] → [offset_t2] → [t2] →
   [current_new_tree_offset] * [new_tree]*)
```

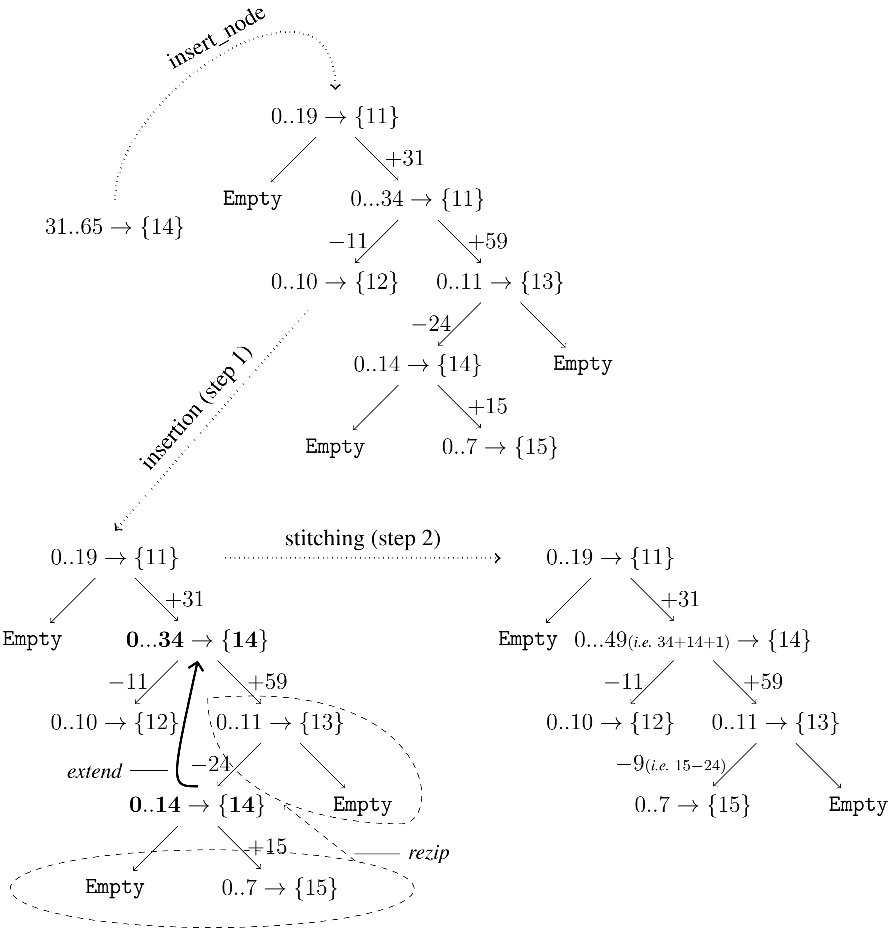

Figure 7 – Inserting a node: stitching illustrated

This function, despite its primary importance, is not unnecessarily complicated to implement. Actually, it makes extensive use of the `make_node` function and of recursive calls to itself. A simplified version can be informally stated as follows, assuming n_1 and n_2 are the current nodes of the two trees t_1 and t_2 to be merged:

1) If $I_{n_1} \cap I_{n_2} = \emptyset$:
Let n_{max} be the highest ranked node between n_1 and n_2 and n_{min} the other one, and t_{max} and t_{min} the respective trees they belong to. Let also $subt^+_{max}$ be the subtree n_{min} should be included in: it is the left subtree of t_{max} if $max(n_{min}) < min(n_{max})$, the right one otherwise. Let $subt^-_{max}$ be the other unchanged subtree of t_{max} .
Merge $subt^+_{max}$ and t_{min} into a new tree t'.
Make a new tree from n_{max}, t' and $subt^-_{max}$.

2) Otherwise, let $I = I_{n_1} \cap I_{n_2}$ and compute the value(s) it contains according to the ones contained in I_{n_1} and I_{n_2}. Let $I^<_{n_1}, I^>_{n_1} = I_{n_1} \backslash I$. These two new intervals represent the lower part of I_{n_1} not in I and the upper part of I_{n_1} not in I. Let similarly $I^<_{n_2}, I^>_{n_2} = I_{n_2} \backslash I$.
Add $I^<_{n_1}$ to the left subtree of t_1 and $I^>_{n_1}$ to the right subtree of t_1. Do the same for $I^<_{n_2}, I^>_{n_2}$ and t_2.
Merge both new left subtrees and merge both new right subtrees.
Make a new smart tree with I and the results of the previous recursive calls.

Note that the values mapped to the intervals are not changed except when the two trees have overlapping intervals.

5.5. *Caching*

As noted in Sect. 4.4, the implemented functions often need to access the rightmost and leftmost bindings of a subtree (i.e. those directly on the right and left-hand side of the current node if the interval is looked at linearly). This is right now naively done by recursively descending the subtree. Another solution would consist in borrowing ideas from monoid caching trees [8] and have fingers [7] pointing at the rightmost left and leftmost right children.

However, this solution was not chosen. We chose to save the two words necessary at each node to record the rightmost and leftmost bindings. In our context, the saved space can be put to a better use by creating caches for high-level operations, even if in the case of a cache miss, the operation takes a little longer because of the logarithmic access to these leftmost and rightmost bindings.

5.6. *Sharing*

The representation of relative subtrees described in Sect. 4.3 allows maximal sharing on subtrees of the data structure: rangemaps are actually DAGs and not trees. The implemented version is on the right-hand side of Fig. 8. Note that `Empty` subtrees are not drawn as shared although they actually are: this comes for free for all nullary constructors in OCaml with no need of any support in the rangemaps implementation.

The trees in Fig. 8 represent shared and unshared rangemaps for the sequential interval: $0..19 \rightarrow \{11\}, 20..29 \rightarrow \{12\}, 30..49 \rightarrow \{11\}, 50..59 \rightarrow \{12\}, 60..100 \rightarrow \{11\}$ with respective ranks $\infty, 3, 5, 3, 6$.

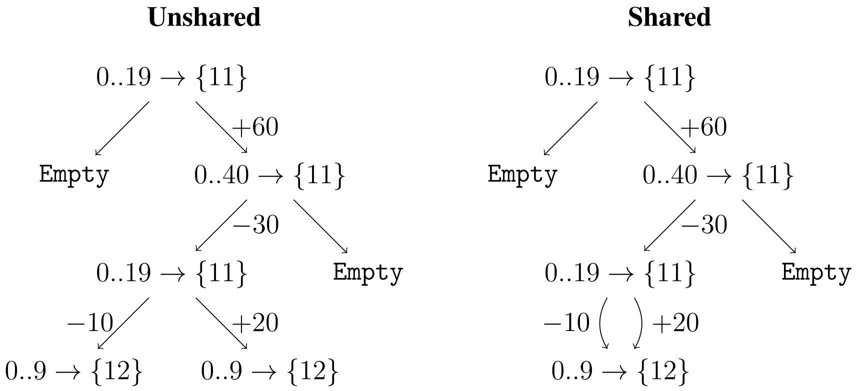

Figure 8 – Unshared vs. shared version of the same tree

The benefit of sharing subtrees can be quite significant. A look at Fig. 9 and Fig. 10 shows the expected space gain when the leaves offer a repeated pattern of similar bindings.

The state of sharing observed in these figures is also a consequence of the fact that nodes of rangemaps are linked relatively to their parents.

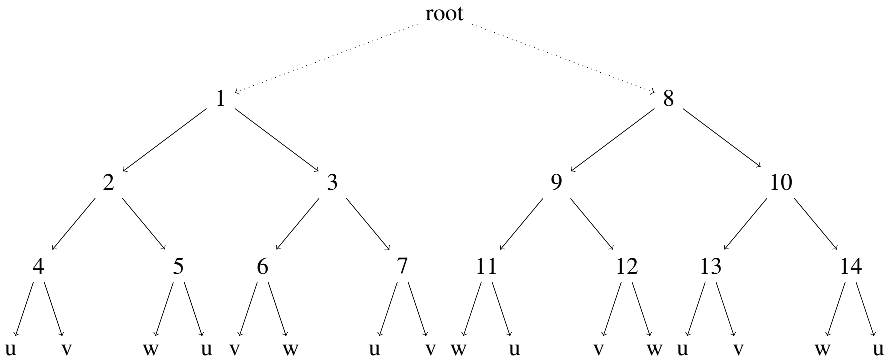

Figure 9 – Unshared subtrees

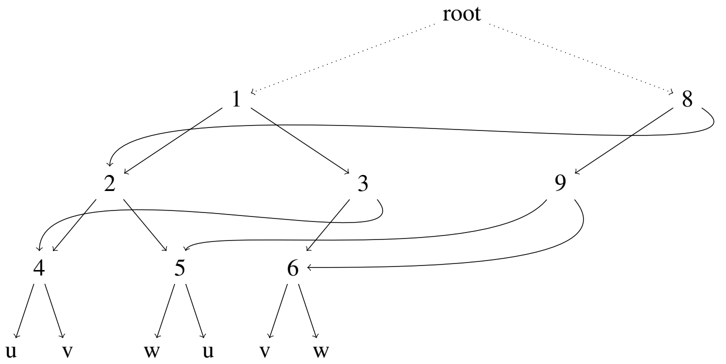

Figure 10 – Shared subtrees

5.7. *Finding one value*

Rangemaps should allow for efficient requests when searching for specific intervals. One difficulty to watch for is that the value to be read may range over more than one interval-key of the rangemap.

More specifically, the find function has the following type:

```
val find: int64 → int64 → t → V.t
(* [offset] → [size] → [tree] → [value] *)
```

Requests are made to read a specific size from a given offset in a tree assumed to be rooted at 0. The bindings found at the requested location may require various amounts of processing in the making of the answer:

– the simplest case happens when the request corresponds exactly to an interval stored in the rangemap: it suffices to return the stored value;

– the request may be for part only of a binding as it exists in the tree: in this case a function provided as a parameter of the data structure is called to extract the relevant bits of the stored value;

– in the most complex case, the request encompasses more than one interval as illustrated in Fig. 11: values extracted from the relevants bindings are stitched together to form a new value;

Actually, the second case is a specific form of the last case, where no real stitching of values is involved.

The code for the `find` function goes as follows. We see the two cases and the call to `extract_bits_and_stitch` which does the work of merging values from the current extracted bits and the accumulator. How the stitching is done actually heavily depends on the kind of values stored in the bindings of a rangemap. From the point of view of a programmer who wants to use the data structure, the functions that need to be provided are those respectively for extracting a part of a binding and to put together a value by stitching together such parts. Also, note that when several bindings contribute to the result, the navigation from one binding to the next relies on `move_right` which uses the zipper returned by `find_bit`.

```
let find offset size tree =
(* locate the node we want *)
let sup_to_read = offset + size - 1 in
let z, cur_off, root = find_bit offset tree in
match root with
(* Not_found has been raised in the Empty case *)
| Empty → assert false

| Node (max, _, _, _, _subr, r, m, v) →
(* We need not further than the current_interval
   AND the alignment wrt to values is okay
*)
```

```
if (Int.le sup_to_read sup_interval) &&
 ... (* some conditions related to values *)
 then v
 else
 (* V.singleton_zero is the neutral element
    for mergeable values
 *)
  let acc = ref V.singleton_zero  in
  let cur_root = ref root in
  let cur_off = ref cur_off in
  let cur_zip = ref z in
  while (Int.le !cur_off sup_to_read) do
  (* extract_bits from current interval
     and stitch it with the current value in
     [acc]
  *)
  acc ← extract_bits_and_stitch
          (* initial [offset] and [size] to read *)
          offset size
          (* offset and node from which to extract bits *)
          !cur_off !cur_root
          (* accumulated merged values up until now *)
          !acc;
   if sup_interval ≥ sup_to_read then
    (* Bogus offset set to biggest integer to end the loop *)
     cur_off ← max_int
   else
    (* Nominal behavior: do next binding *)
    begin
     let o, t, z = move_right !cur_off !cur_root !cur_zip in
     cur_root ← t;
     cur_zip ← z;
     cur_off ← o;
    end
  done;
  !acc
;;
```

5.8. *Caching calls about finding values*

When used in the value analysis, one major cause for performance degradation in lookups is when the location to be read is itself not known with precision. It could for instance be the whole range of the indices of an array, or the addresses of all occurrences of a specific field in a struct array.

It is this higher-level query for a set of indices that rangemaps have been designed to be able to optimize through caching. Consider the analysis of, say, a loop scanning an array. At each iteration in the search

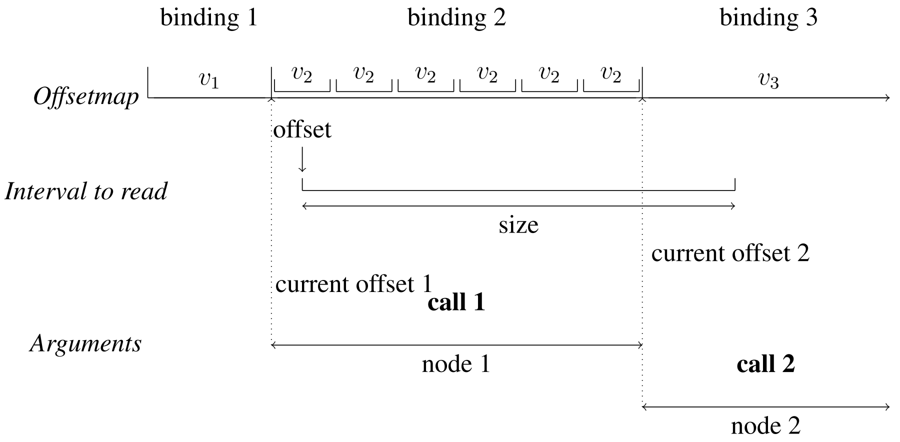

Figure 11 – Construction of new values ranging over multiple intervals

for the loop's fixpoint, similar (but not necessarily identical) sets of indices are used in requests to similar (but not necessarily identical) rangemaps. Large sets of indices are represented as intervals with periodicity information in order to distinguish requests for all occurrences of a specific field in an array of structs from requests for all values from the array. This situation occurs commonly in analyzed programs.

We should avoid caching too specific information, for instance specific to one requested set of indices, or to one rangemap. The key of a cache entry is a pair made from a subtree and a subrequest. Our proposal is to cache only requests for any kind of specific periodicity information, reading from the entire range of a subtree. It seems to us that these are the entries that are most likely to be reused.

6. Conclusion and further work

In this article, we have introduced rangemaps, a data structure to represent persistent maps indexed by intervals. We have shown the unique properties of rangemaps and highlighted details of the current prototype implementation.

The next step is to replace the existing ad-hoc data structure for representing char arrays in Frama-C's value analysis [1] by rangemaps. We expect this will increase the efficiency of the value analysis both in terms of memory used and speed.

The first preliminary (internal and unpublished) results also show an interesting improvement on the accuracy of the analysis. We are looking forward to the possibility to run benchmarks comparing more deeply the current ad-hoc representation and the representation based on rangemaps.

Acknowledgements

We would like to thank the reviewers of this article for their detailed and very constructive comments. They have really helped us improve this paper.

References

[1] Georgii M. Adelson-Velskii and Evgenii M. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3:1259–1263, 1962.

[2] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[3] Sylvain Conchon and Jean-Christophe Filliâtre. Type-Safe Modular Hash-Consing. In *ACM SIGPLAN Workshop on ML*, Portland, Oregon, September 2006.

[4] Pascal Cuoq and Damien Doligez. Hashconsing in an incrementally garbage-collected system: a story of weak pointers and hashconsing in OCaml 3.10.2. In *ML '08: Proceedings of the 2008 ACM SIGPLAN workshop on ML*, pages 13–22, New York, NY, USA, 2008. ACM.

[5] Martin Erwig. Diets for fat sets. *J. Funct. Program.*, 8(6):627–632, 1998.

[1] The simplest way to handle heterogeneous pointer casts in an abstract interpretation-based static analyzer is to decide from the start to treat every memory location as a char array. In fact, the value analysis treats every memory location as a bit array as part of its treatment for bitfields in C.

[6] The Frama-C development team. Frama-C. `http://frama-c.cea.fr`.

[7] Leonidas J. Guibas, Edward M. McCreight, Michael F. Plass, and Janet R. Roberts. A new representation for linear lists. In *STOC*, pages 49–60. ACM, 1977.

[8] Ralf Hinze and Ross Paterson. Finger trees: a simple general-purpose data structure. *J. Funct. Program.*, 16(2):197–217, 2006.

[9] Gérard Huet. The zipper. *J. Funct. Program.*, 7(5):549–554, 1997.

[10] David Monniaux. Verification of device drivers and intelligent controllers: a case study. In Christoph Kirsch and Reinhard Wilhelm, editors, *EMSOFT*, pages 30–36. ACM & IEEE, 2007.

[11] Donald R. Morrison. PATRICIA - Practical Algorithm To Retrieve Information Coded In Alphanumeric. *J. ACM*, 15(4):514–534, 1968.

[12] Chris Okasaki and Andy Gill. Fast mergeable integer maps. In *Workshop on ML*, pages 77–86, 1998.