# An Incremental Heap Canonicalization Algorithm

Madanlal Musuvathi[1] and David L. Dill[2]

[1] Microsoft Research, Redmond, madanm@microsoft.com
[2] Computer Systems Laboratory, Stanford University, dill@cs.stanford.edu

**Abstract.** The most expensive operation in explicit state model checking is the hash computation required to store the explored states in a hash table. One way to reduce this computation is to compute the hash incrementally by only processing those portions of the state that are modified in a transition. This paper presents an *incremental* heap canonicalization algorithm that aids in such an incremental hash computation. Like existing heap canonicalization algorithms, the incremental algorithm reduces the state space explored by detecting heap symmetries. On the other hand, the algorithm ensures that for small changes in the heap the resulting canonical representations differ only by relatively small amounts. This reduces the amount of hash computation a model checker has to perform after every transition, resulting in significant speedup of state space exploration. This paper describes the algorithm and its implementation in two explicit state model checkers, CMC and Zing.

## 1 Introduction

There is a practical need to apply verification techniques to large software systems. In this vein, explicit state model checkers that systematically enumerate the possible states of a given system have been successful in finding complex errors, and sometimes proving their absence in software systems [1–4].

Apart from the well known state explosion problem, one challenge in scaling explicit state model checkers to large systems is the sheer size of individual system states. These model checkers store the explored states in a hash table to avoid exploring them redundantly. The hash computation required in this process is the most time consuming operation during model checking [5]. At the minimum, computing a hash value requires a few arithmetic operations for *every* byte of the state. This can be very expensive, especially for states that are tens or hundreds of kilobytes.

One way to reduce the hash computation overhead is to compute the hash *incrementally.* During model checking, a transition is very likely to modify only small portions of the state. By accounting for these differences between the initial and final state of a transition, and using a suitable hash function, a model checker can generate the hash value of the final state by incrementally updating the hash value of the initial state. (See §2.3 for details.) This amortization of hash computation over the unmodified portion of the state can significantly improve the speed of state space exploration.

Implementing such an incremental hashing scheme is complicated by the need for software model checkers to perform *heap canonicalization* [6, 7]. Almost all non-trivial programs allocate memory in the heap. Heap canonicalization is a state space reduction technique that enables model checkers to identify states that are behaviorally equivalent but differ only in the memory locations of heap objects. To identify such equivalent states, a model checker executes a heap canonicalization algorithm to transform a state to a canonical representation that is unique for all equivalent states. This canonical representation is then inserted in the hash table.

Unfortunately, the only known heap canonicalization algorithm [7] does not admit incremental processing. This algorithm performs a depth first traversal of the heap, and the canonical representation of each object depends on its depth first ordering. As a result, even small structural changes to the heap, such as object additions or deletions can modify the canonicalization of a large number of objects in the heap. Thus, even when a transition modifies small portions of the state, the canonical representations of the initial and final states can be significantly different. This forces the model checker to process large portions of the state during hash computation.

This paper presents an improved, *incremental* heap canonicalization algorithm. Like [7], this algorithm generates a unique canonical representation for all equivalent heap states. However, this algorithm ensures that small changes to the heap only result in relatively small changes in the canonical representation. This allows the model checker to perform incremental hash computation. The basic idea of the incremental algorithm is to determine the canonicalization of a heap object from its *shortest path* to some global variable. When a transition makes small changes to the heap structure, the shortest path of most objects is likely to remain the same [8, 9]. A model checker only needs to process those objects whose shortest paths have changed in a transition.

We have implemented this incremental algorithm in two explicit state model checkers, CMC [4] and Zing [10]. The algorithm is very easy to implement, and can handle arbitrary data structures in the heap, including type-unsafe pointers. We have applied the algorithm for several large models and have achieved an improved performance in all of them (§5). For the examples we have tried the model checker processes at most 5% of the objects in the heap during hash computation using the incremental heap canonicalization algorithm. This results in a model checking speedup of 2 to 9 times. While providing this speedup, the incremental algorithm preserves the state space reduction provided by the previous heap canonicalization algorithm [7].

## 2 Preliminaries

This section describes the necessary formalisms required to explain the main ideas in the paper. Motivated by our efforts in checking C programs, we will not assume that the heap pointers are *type-safe*. Specifically, we will allow a pointer variable to point to objects of different types at different instances in the program. Also, we will allow these variables to point to arbitrary fields *in* the object.

### 2.1 Heap Objects

At any instant, the program state includes a collection of heap objects occupying memory addresses from a countably infinite set $H$. To allow arbitrary pointer arithmetic, we define the normal arithmetic operations, such as $+$, $\leq$, over $H$ in the obvious way.

A heap object is identified by its start address $\in H$. For an object $p$, $len(p)$ represents the length of the object. The fields of an object $p$ of length $l$ occupy memory locations in the range $[p, p+l)$. For simplicity, assume that a field of an object $p$ is named by its integer offset $f$ in $p$, where $0 \leq f < len(p)$. The term $p[f]$ denotes the value of a field $f$ in $p$, where the value $p[f]$ is either an integer, or a pointer value $\in H$, or the special *null* pointer. An object $p$ *points to* another object $q$ exactly when there is a field $f$ of $p$ such that $q \leq p[f] < q + len(q)$. In this case, $p$ is said to contain a *pointer* to $q$.

### 2.2 Program State

Apart from the heap, the state of a program at any instant consists of all the global variables and the stacks of all the threads. We seek to capture the entire state of the program in the heap as follows.

Conceptually, all the global variables in a program can be considered as fields of a global *root* object that represents the statically allocated region in memory. For ease of exposition, assume that the addresses of all global variables are in $H$. Also, the stack of each thread is represented as a linked-list of stack frames, where each stack frame is a heap object. We assume that for each thread there is a field in *root* that points to its stack. [3]

The program state $S$ is a set of objects $\{root = p_0, p_1, p_2, \ldots, p_n\}$. Given a state $S$, we assume that all objects in $S$ do not overlap in memory. In other words, for any $p_i, p_j \in S, p_i \neq p_j \Rightarrow p_i < p_j \vee p_i \geq p_j + len(p_j)$. Also, we assume that all objects in a state are *reachable* from the *root* object. That is, for any object $p \in S$, there is a sequence $\langle root = p_0, p_1, \ldots, p_n = p \rangle$ such that $p_i$ points to $p_{i+1}$ for $0 \leq i < n$. We assume that any object that is not reachable from the *root* object is automatically removed from the state. [4]

[3] We also assume that each thread is *statically* identified by a unique identifier. Detecting *thread-symmetries* by permuting these identifiers is an interesting problem, but beyond the scope of this paper.

[4] Such memory-leak detection (or garbage collection) can be done during the canonicalization algorithm described in Section 4.

### 2.3 Incremental Hashing of State

The performance bottleneck in an explicit model checker is the hash computation required when storing the state in a hash table. One way to mitigate the performance overhead is to compute the hash *incrementally* as follows.

For a given state, the model checker computes a *partial* hash value for each object in the state. The hash value of the entire state is obtained from these partial hash values. The goal is to cache these partial hash values with objects, and recompute them only when a transition explored by the model checker modifies the object. However, if an object remains unmodified in a transition, its cached value can be used when computing the hash value of the final state. This amortization of hash computation across states improves the model checking performance.

Formally, we assume the existence of three hash functions $h_o$, $h_S$, and $h_u$ defined as follows. For an object $p$ of length $l$, there is a hash function $h_o$ that takes $l+1$ arguments. The partial hash value of $p$ is given by

$$\mathcal{H}(p) = h_o(p, p[0], p[1], \ldots, p[l-1])$$

Note, to minimize hash collisions any *good* hash function should use values of all the fields in an object. Also, to avoid collisions between objects whose values are permutations of each other, the hash function should use the location of the object in the state. For a state $S = p_0, p_1, \ldots, p_n$, there is a hash function $h_S$ that generates the hash for the state using the partial hash values of the objects.

$$\mathcal{H}(S) = h_S(\mathcal{H}(p_0), \mathcal{H}(p_1), \ldots, \mathcal{H}(p_n))$$

Finally, we assume that $\mathcal{H}(S)$ can be incrementally updated. Let $S(p, p')$ represent the state obtained from $S$ by modifying the object $p \in S$ by $p'$ and leaving all other objects unmodified. One of $p$ or $p'$ can be *null* to represent object allocations and deletions respectively. Given $S' = S(p, p')$ assume there is a hash update function $h_u$ to obtain the hash value of $S'$.

$$\mathcal{H}(S(p, p')) = h_u(\mathcal{H}(S), \mathcal{H}(p), \mathcal{H}(p'))$$

Typically, the update function $h_u$ is computationally much more efficient than $h_o$ or $h_S$. After a transition, a model checker applies $h_o$ on all object modified in a transition and then uses $h_u$ once for each modified object to determine the hash value of the final state. In effect, the cost of computing the hash value is proportional to the total size of the objects modified in a transition.

In practice, it is very straightforward to design hash functions $h_o, h_S, h_u$ from existing hash functions. See Appendix B for such an example.

The main goal of this paper is to allow this incremental hash computation when model checkers perform heap canonicalization.

## 3 Heap Canonicalization

This section describes the need for heap canonicalization when model checking software programs and presents an informal description of the Iosif's algorithm [7].

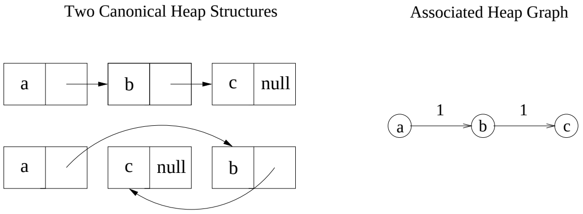

**Fig. 1.** Two equivalent representations of a linked list, and their common heap graph. This first element of the linked list is the global node.

When a program dynamically allocates objects in the heap, the exact memory location of these objects is arbitrary from the perspective of the program. The memory location of an object is determined by some internal heap allocation algorithm and typically depends on the order of all previous object allocations and deletions.

When a model checker explores different event interleavings during state exploration, it can generate multiple representations of the heap that differ in the memory locations of the objects, but are otherwise equivalent. For example, Figure 1 shows two representations of a linked list with three elements. The two representations differ in the memory locations of the second and the third element in the list. As heap objects can be accessed only through pointers from global variables or other heap objects, no program can differentiate between the two different representations in Figure 1. [5] From the perspective of the model checker, these two representations describe the same state, and thus should explore at most one of them.

### 3.1 The Heap Graph

To formalize the notion of the equivalence of heap states, one can introduce the notion of a heap graph. Informally, a heap graph *abstracts* the memory locations of the heap objects, while maintaining information about the pointers to these objects.

Given a state $S$, the heap graph $G(S)$ is defined as follows. For each object $p \in S$, $G(S)$ contains a vertex given by $V(p)$. There is a directed edge in $G(S)$ from $V(p)$ to $V(q)$ if and only if $p$ points to $q$ in $S$. This edge is labeled by the offset of the field in $p$ that contains the pointer to $q$. For instance, the heap graph of the linked list is shown in Figure 1. Obviously, no two outgoing edges of an object have the same label. Each node in the heap graph is labeled by the values of all the non-pointer fields in the corresponding object as shown in Figure 1.

[5] Even in C programs, which are allowed to inspect the value contained in pointers, any behavior that relies on absolute values of the pointers can be safely marked as an error.

**Propostion 1** *A program cannot differentiate between two states $S_1$ and $S_2$ if their heap graphs $G(S_1)$ and $G(S_2)$ are isomorphic.*

Basically, a heap graph captures the entire state of the heap along with all the global variables, but abstracts the memory addresses contained in the pointer variables.

### 3.2 The Canonical Representation of a Heap

The aim of a heap canonicalization algorithm is to produce the same canonical representation for all heaps that have the same heap graph. By computing the hash on this canonical representation, the model checker can ensure that it explores at most one among the possible equivalent states.

The generation of a canonical representation can be considered as a *relocation* of the objects in the heap. Conceptually, the canonicalization algorithm relocates each object to a *canonical* location determined by the algorithm. After relocation, the algorithm modifies all pointers to an object to reflect this new location.

Formally, a canonicalization algorithm defines a relocation function $reloc : H \rightarrow H$ such that $reloc(p)$ determines the canonical location of an object $p \in S$. We restrict the *reloc* function to have the following desirable properties. First, it is not necessary to relocate the *root* object. Thus,

$$reloc(root) = r, \text{ for a constant } r \tag{1}$$

Second, it is desirable to relocate heap objects in their entirety. In other words, the offset of fields in an object should be invariant during relocation.

$$reloc(p + f) = reloc(p) + f, \quad \forall f : 0 \leq f < len(p) \tag{2}$$

Equations 1 and 2 imply that a particular heap canonicalization algorithm only defines the relocation function for start addresses of heap objects. The two equations determine the relocation for other addresses $\in H$.

Finally, there is a constraint on the relocation function defined by any heap canonicalization algorithm. The *reloc* function should not overlap objects in the canonical heap. That is, for all $p, q \in H$

$$reloc(p) = reloc(q) \iff p = q \tag{3}$$

Given a relocation function that satisfies the above constraints, the canonical representation of a state can be obtained as follows. First, relocate each object $p \in S$ to the location $reloc(p)$. Also, if an object $p$ points to $q$, modify the pointer value to reflect the new location of $q$. The values of all non-pointer fields do not change in this relocation. That is,

$$reloc(p)[f] = \begin{cases} reloc(p[f]) & \text{if } p[f] \in H \\ p[f] & \text{otherwise} \end{cases}$$

Abusing notation we will define $reloc(S)$ as the state thus obtained from $S$.

The goal of a heap canonicalization algorithm is define a relocation function such that $reloc(S) = reloc(S')$ whenever the heap graphs $G(S)$ and $G(S')$ are isomorphic. Given a state $S$, a model checker computes $reloc(S')$ and inserts the latter in the hash table. Note, it is not necessary to *physically* relocate the objects in the heap. The model checker merely applies the *reloc* function on all pointer values in the heap when computing the hash.

### 3.3 Iosif's Algorithm

Iosif's algorithm [7] involves a depth first traversal of the heap graph starting from the *root* object. The traversal uses the edge labels in the heap graph to deterministically order all outgoing edges of a node. [6] The algorithm relocates the heap objects in the heap in the order visited by the traversal. Specifically, if $p_i$ represents the object with depth first order number $i$, then

$$reloc(p_i) = \sum_{j=0}^{i-1} len(p_j) \tag{4}$$

Note that the above relocation function satisfies the constraint in Equation 3.

Figure 2(a) shows an example. The heap consists of three objects in a binary tree. The *head* node is the *root* object and contains pointers to a *left* node and a *right* node. Assuming that the size of these nodes is 3 word lengths, the algorithm relocates the head, left and right nodes at offsets 0, 3 and 6 respectively in the canonical heap. Also, after relocating the objects, the algorithm modifies the left and right pointers in the head node to point to the respective objects in their new locations.

The correctness of the Iosif's algorithm follows. If two heap states have the same heap graph, then the algorithm will visit the heap objects in the same order and relocate them at the same locations. This produces the same canonical representation.

### 3.4 Need for an Incremental Algorithm

Now we can see why Iosif's algorithm is not suitable for incremental hash computation — it unnecessarily modifies the relocation of objects in the canonical heap. To illustrate this, consider the heap in Figure 2(a) and a transition that deletes the left node in the binary tree. Figure 2(b) shows the resulting heap and the canonical representation generated by the Iosif's algorithm. The algorithm locates the right node at offset 3 in the canonical heap, while the node had an offset 6 before the transition. This change in the location invalidates the precomputed hash value for the right node, even though the node was not modified in the current transition.

[6] Without such an ordering, the heap canonicalization problem is an instance of the graph isomorphism problem, and so is intractable [11].

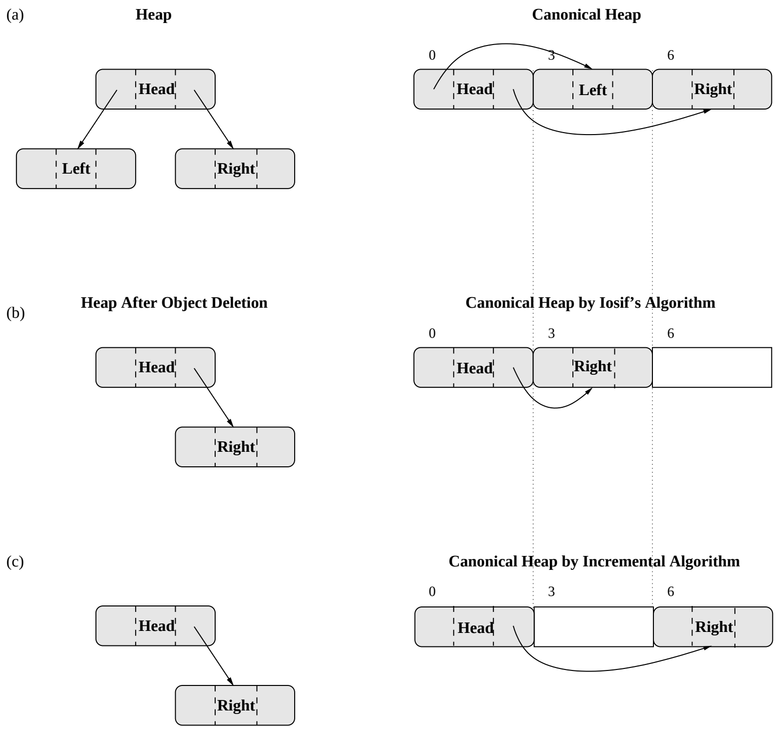
(a)
Heap
Head
Left
Right
Canonical Heap
0
3
6
Head
Left
Right
(b)
Heap After Object Deletion
Head
Right
Canonical Heap by Iosif's Algorithm
0
3
6
Head
Right
(c)
Head
Right
Canonical Heap by Incremental Algorithm
0
3
6
Head
Right

**Fig. 2.** Demonstration of Heap Canonicalization before and after a transition that deletes a node. The head node is global.

In general, Iosif's algorithm determines $reloc(p)$ from the depth first order number of $p$ in the heap graph (see Equation 4). Any change in the heap graph can potentially modify the relocation of all objects after the change in the depth first order. For instance, one can expect an object addition or deletion to modify on average the relocation of half of all the heap objects. Moreover, whenever $reloc(p)$ changes, the algorithm needs to modify all the pointers to $p$. This invalidates the precomputed hash for any object that contains such a pointer. In practice, the Iosif's algorithm requires recomputing the hash value for large portions of the heap, drastically slowing the model checker performance.

## 4 Incremental Heap Canonicalization

This section describes an incremental heap canonicalization algorithm that improves upon the Iosif's algorithm.

The incremental algorithm has two requirements. First and foremost, the algorithm should guarantee heap canonicalization, and thus generate the same canonical heap for all equivalent heaps. Second, the algorithm should seek to reduce unnecessary changes to the canonical representation for small changes in the heap. Ideally, after every transition the model checker should only need to recompute the hash for objects that are modified in the transition.

To illustrate this, consider the example in Figure 2. When the transition deletes the left node, the incremental algorithm generates a canonical heap as shown in Figure 2(c). Note, the right node remains at offset 6 in the canonical heap both before and after the transition. This enables the model checker to reuse the hash computed for the object before the transition. To guarantee canonicalization, the incremental algorithm should produce the canonical heap in Figure 2(c) for any heap equivalent to the state in Figure 2(c). It is important to note once again that the canonical heap is a conceptual representation used during hash computation – the heap itself is not physically modified. In particular, the empty space between the two objects in Figure 2(c) does *not* represent unreclaimed space in the heap.

The basic idea behind the incremental algorithm is to determine the relocation of an object from the *bfs access chain* of the object, which is a shortest path between the object and the global object. For example, the right node in Figure 2 always has the same bfs access chain that consists of the right pointer from the head node. Thus, the incremental algorithm *always* relocates the right node at the same offset in the canonical heap, irrespective of other objects in the heap. On small changes to a graph, the shortest paths between most objects are likely to remain the same [8, 9]. This is specifically so for the heap graph of programs, which typically use large number of data structures that are only weakly related.

### 4.1 Access Chains

Heap objects can only be accessed through pointers from global variables. For instance, a C program can access the right node in Figure 2(a) with an expression `head->right`. In general, a heap object $p$ can be accessed through a chain of pointers. This access can be represented by the sequence $root = p_0, f_0, p_1, f_1, \ldots, p_n = p$, where $f_i$ is the offset of a field in $p_i$ that contains a pointer to $p_{i+1}$, for $0 \leq i < n$. In the heap graph, this chain forms a path starting from *root*. This path is uniquely defined by the offset labels on the path edges.

Formally, an *access chain* of a heap object $p$ is a path in the heap graph from the *root* object to $p$ and is denoted by $\langle f_0, f_1, \ldots, f_{n-1} \rangle$, the list of offset labels on the path edges. For instance, the access chain of the third element in the linked list of Figure 1 is $\langle 1, 1 \rangle$, assuming that the first element is the *root* object. Also, the *root* object has an empty access chain $\langle \rangle$.

### 4.2 BFS Access Chain

A breadth first traversal of the heap graph naturally defines an access chain for objects in the graph. During a breadth first traversal, the edges used to traverse the graph form a spanning tree of the graph rooted at the global node. For any object in the graph, this spanning tree provides a shortest path from the global node to that object. Obviously, the access chain that corresponds to this path is one of the shortest of all access chains for the object. Additionally, if the breadth first traversal traverses the edges from an object in the increasing order of their offset labels, then the access chain constructed above is guaranteed to be the lexicographically smallest of all shortest access chains of the object.

Formally, a *bfs access chain* of an object $p$ given by $\langle p \rangle$, is the access chain $\langle f_0, f_1, \ldots, f_n \rangle$ such that for any other access chain $\langle g_0, g_1, \ldots, g_m \rangle$ of the object the following holds: either $m > n$; or $m = n$ and there is an $i < n$ such that for all $0 \leq j < i$, $f_j = g_j$ and $f_i < g_i$. The bfs access chain of all objects in the heap graph can be constructed by performing a breadth first traversal of the graph that traverses all outgoing edges of a node in the edge label order. Given a heap graph, the *bfs access chain* of an object is unique. The incremental heap canonicalization algorithm uses this chain to determine the location of objects in the canonical heap.

### 4.3 Defining the *reloc* Function

A heap canonicalization algorithm is defined by the *reloc* function that determines the location of a heap object $p$ in the canonical representation. The aim of the incremental algorithm is to define *reloc* such that $\mathit{reloc}(p)$ depends *only* on $\langle p \rangle$.

Also, in a type unsafe language such as C, a single pointer field can point to objects of multiple types, and hence multiple lengths. Accordingly, objects of different lengths can have the same bfs access chain. To differentiate such objects in the canonical representation, we also require that $\mathit{reloc}(p)$ depends on $\mathit{len}(p)$. This, as will be explained below, is also crucial to satisfy Equation 3.

In summary, the incremental algorithm seeks for a function *canon* that takes $\langle p \rangle$ in a suitable encoding and $\mathit{len}(p)$ as arguments, and returns $\mathit{reloc}(p)$.

$$\mathit{reloc}(p) \equiv \mathit{canon}(\langle p \rangle,\ \mathit{len}(p)) \tag{5}$$

The next task is to obtain an encoding for $\langle p \rangle$. Let $\langle p \rangle = \langle f_0, f_1, \ldots, f_n \rangle$. If $\mathit{parent}(p)$ is the parent of the object $p$ in the breadth first traversal of the heap graph, then $\langle \mathit{parent}(p) \rangle = \langle f_0, f_1, \ldots, f_{n-1} \rangle$. This hints at an encoding for $\langle p \rangle$ from the encoding for $\langle \mathit{parent}(p) \rangle$ and $f_n$. Using Equation 5 recursively, the following theorem shows that $\mathit{reloc}(\mathit{parent}(p) + f_n)$ provides the necessary encoding for $\langle p \rangle$.

**Theorem 1.** *For any arbitrary function canon* $: H \times N \rightarrow H$*, the relocation function defined by*

$$\mathit{reloc}(p) = \mathit{canon}(\mathit{reloc}(\mathit{parent}(p) + f_n),\ \mathit{len}(p)) \tag{6}$$

*relocates two objects with the same bfs access chain and the same length to the same location in the canonical heap.*

**Proof:** The proof follows from a simple induction on the depth of the bfs access chain of an object. The global object has an empty bfs access chain and the base step trivially follows from Equation 1. Assume the theorem holds for all objects with a bfs access chain of length $\leq n$. Consider two objects $p_1$, $p_2$ such that they have the same length and the same bfs access chain. Let

$$\langle p_1 \rangle = \langle p_2 \rangle = \langle f_0, f_1, \ldots, f_n \rangle$$

Obviously, $parent(p_1)$ and $parent(p_2)$ have the same bfs access chain of length $n$.

$$\langle parent(p_1) \rangle = \langle parent(p_2) \rangle = \langle f_0, f_1, \ldots, f_{n-1} \rangle$$

By induction

$$reloc(parent(p_1)) = reloc(parent(p_2))$$

Using Equation 2,

$$reloc(parent(p_1) + f_n) = reloc(parent(p_2) + f_n)$$

This from Equation 6 and the fact that $len(p_1) = len(p_2)$ implies that

$$reloc(p_1) = reloc(p_2)$$

□

Note that $f_n$ is the offset of the field in $parent(p)$ that points to $p$, and thus $reloc(parent(p) + f_n)$ represents the address of that field in the canonical heap. Theorem 1 essentially shows that $\langle p \rangle$ can be captured by this address in the canonical heap. This greatly simplifies the implementation of the incremental canonicalization algorithm.

The following theorem follows.

**Theorem 2.** *For an arbitrary function canon* $: H \times N \rightarrow H$*, the relocation function defined by Equation 6 generates the same canonical representation for two heaps with the same heap graph, provided Equation 3 holds.*

**Proof:** The bfs access chain is an inherent property of an object in the heap graph. Thus, given two heaps with the same heap graph, equivalent objects in the two heaps have the same bfs access chain. The proof follows from Theorem 1. □

### 4.4 Designing the *canon* function

By Theorem 2, heap canonicalization is guaranteed for *any* function *canon* $: H \times N \rightarrow H$ that satisfies the constraint in Equation 3. The final task is to design one such function.

Without apriori knowing the length of the objects with different bfs access chains, a closed form for the *canon* function is not possible. The trick then, is to define the *canon* function *incrementally* during model checking. The algorithm is shown in Figure 3 and implements *canon* as a hash table. Initially, the hash table is empty. When the value for $canon(addr, len)$ is required for a new address length pair, the algorithm *allocates* a new region in the canonical heap of the required length. This region is never reused for other address length pairs. Thus no overlap is possible in the canonical heap, satisfying Equation 3.

The *canon* hash table is a global table maintained by the model checker. During state exploration, the model checker can add new entries to the table, but can never modify an existing entry or delete it. This ensures that the hash table implements a *function* which is essential from Theorem 2 for heap canonicalization. Also, the size of the *canon* table grows as the model checker discovers different bfs access chains in the program. However, the number of such chains tends to stabilize once the data structures are initialized in the program. In our experiments (§5), the *canon* table did not exceed ten thousand entries.

As a demonstration of the algorithm, consider the example in Figure 2. After processing the state in Figure 2(a), the *canon* table maintains the following mapping: $(0, 3) \rightarrow 3$ and $(2, 3) \rightarrow 6$. In other words, the *canon* table remembers that an object of length 3 pointed from the offset 2 (the right pointer) of the head node should be relocated at offset 6 in the canonical heap. Now, when the left node is deleted, the mapping maintained above produces a canonical representation in Figure 2(c) as desired.

## 5 Experimental Results

We have evaluated the incremental heap canonicalization algorithm in two explicit model checkers, CMC [12] and Zing [10]. CMC is specifically designed for checking network protocol implementations. It executes the implementation directly without resorting to any intermediate representation. A transition involves the entire processing of a protocol event, such as packet receives or timer interrupts, and can typically involve more than tens of thousands of instructions. Zing focuses on detecting concurrency errors in large software programs. The input language is designed for automatic translation of software programs into Zing models, and provides support for dynamic object and thread creation. Zing explores thread interleavings at much finer granularity than CMC, and a transition in Zing typically involves few instructions.

CMC and Zing represent two fundamentally different model checkers, and the incremental heap canonicalization algorithm performs well in both of them. Table 1 shows the improvement achieved by the model checkers on three large models. As the state sizes grow, the incremental algorithm fares much better than the non-incremental version. In these examples, the model checker processes only 5% of the heap per transition. As a result, the model checker runs 2 to 9 times faster.

In the Linux TCP case, the incremental algorithm almost always processes only the objects that are modified by the system. However, this is not the case with the Zing models. We believe that this is due to a deficiency of the algorithm implementation in Zing. Specifically, the algorithm does not process the thread stack frames incrementally. We hope to rectify this very soon.

| Model | State<br>Size (KB) | No. of Heap<br>Objects (avg) | % Modified<br>per Transition | % Accessed<br>for Hashing | Model Checker<br>Speedup |
|---|---|---|---|---|---|
| Tx. Manager (Zing) | 1.3 | 46.0 | 0.16 | 4.25 | x2.15 |
| File System (Zing) | 4.1 | 364.1 | 0.4 | 2.18 | x3.59 |
| Linux TCP (CMC) | 255.7 | 102.7 | 4.96 | 5.06 | x8.85 |

**Table 1.** Performance improvement of the Incremental Canonicalization Algorithm

While running our experiments, we found that both CMC and Zing spend their time in performing the breadth-first traversal of the heap required for the incremental algorithm, and *not* on the hash computation. However, note that such a full-heap traversal is required even to check for memory leaks (or to perform garbage collection). One way to avoid this full-heap traversal is to implement an incremental shortest path algorithm, such as [13], for computing the bfs access chains and for detecting memory leaks. We hope to pursue this approach in our future work.

## 6 Related Work

The work presented in this paper is related to, and in many ways relies on the observations made in [6, 7]. The need for heap canonicalization was first observed by Lerda et.al. [6]. However, they only provide a heuristic algorithm that does not guarantee uniqueness of the canonical representation for all equivalent heaps. Iosif's algorithm [7] is the only previously known heap canonicalization algorithm to guarantee this uniqueness. This algorithm has been extended to support thread symmetries [14]. Extending the incremental heap canonicalization algorithm to recognize thread symmetries is interesting future work.

Other researchers have observed the expensiveness of state hashing in explicit model checking [5]. In [15], Dillinger et.al. provide a method to reduce the *number* of hash functions required for bitstate hashing. This approach is orthogonal to the approach presented in the paper and can be used together.

## 7 Conclusions

This paper presents an incremental heap canonicalization algorithm that is necessary when explicitly model checking large software programs. As demonstrated by the experiments, the incremental algorithm scales well to large heaps by generating the canonical representation of a heap incrementally.

## References

1. Holzmann, G.J.: From code to models, Newcastle upon Tyne, U.K. (2001) 3–10
2. Corbett, J., Dwyer, M., Hatcliff, J., Laubach, S., Pasareanu, C., Robby, Zheng, H.: Bandera: Extracting finite-state models from java source code. In: ICSE 2000. (2000)

3. Brat, G., Havelund, K., Park, S., Visser, W.: Model checking programs. In: IEEE International Conference on Automated Software Engineering (ASE). (2000)
4. Musuvathi, M., Park, D., Chou, A., Engler, D.R., Dill, D.L.: CMC: A Pragmatic Approach to Model Checking Real Code. In: Proceedings of the Fifth Symposium on Operating Systems Design and Implementation. (2002)
5. Holzmann, G.J.: The Spin Model Checker: Primer and Reference Manual. Addison Wesley, Boston, Massachusetts (2003)
6. Lerda, F., Visser, W.: Addressing dynamic issues of program model checking. Lecture Notes in Computer Science **2057** (2001) 80–102
7. Iosif, R.: Exploiting Heap Symmetries in Explicit-State Model Checking of Software. In: Proceedings of 16th IEEE Conference on Automated Software Engineering. (2001)
8. Frigioni, D., Marchetti-Spaccamela, A., Nanni, U.: Incremental algorithms for single-source shortest path trees. In: Proceedings of Foundations of Software Technology and Theoretical Computer Science. (1994) 112–224
9. Narvaez, P., Siu, K.Y., Tzeng, H.Y.: New dynamic SPT algorithm based on a ball-and-string model. In: INFOCOM (2). (1999) 973–981
10. Andrews, T., Qadeer, S., Rehof, J., Rajamani, S.K., Xie, Y.: Zing: Exploiting program structure for model checking concurrent software. In: Proceedings of the 15th International Conference on Concurrency Theory. (2004)
11. Gary, M.R., Johnson, D.S. In: Computers and Intractability. Freeman (1979)
12. Musuvathi, M., Park, D., Chou, A., Engler, D., Dill, D.: CMC: A pragmatic approach to model checking real code. In: Proceedings of Operating Systems Design and Implementation (OSDI). (2002)
13. Ramalingam, G., Reps, T.W.: An incremental algorithm for a generalization of the shortest-path problem. J. Algorithms **21** (1996) 267–305
14. Robby, Dwyer, M.B., Hatcliff, J., Iosif, R.: Space-reduction strategies for model checking dynamic software. In: SoftMC03 Workshop on Software Model Checking, Electronic Notes in Theoretical Computer Science. Volume 89. (2003)
15. Dillinger, P.C., Manolios, P.: Bloom filters in probabilistic verification. In: Formal Methods in Computer-Aided Design (FMCAD). (2004)
16. Carter, J.L., Wegman, M.N.: Universal classes of hash functions. (In: Journal of Computing and System Sciences) 143
17. Cormen, T.H., Leiserson, C.L., Rivest, R.L. In: Introduction to Algorithms. MIT Press (1990)

## A Implementation of the *canon* function

See Figure 3

## B Example of an Incremental Hash Function

One example that is shown below is the universal hash function [16] described in [17]. Given a state $x$ consisting of $n$ bytes $x = \{x_1, \ldots, x_n\}$, a hash table of size $m$ where $m$ is prime, and a random array $r$ of $n$ elements $r = \{r_1, \ldots, r_n\}$ such that $0 \leq r_i \leq m - 1$, the hash function is given by

$$h(x) = \sum_{i=1}^{n} r_i x_i \;\; mod\ m$$

```
canon_table; //implemented as a hash table
unalloc_address = canonical_heap_address_start;

canon(parent_ptr, len){
  if (canon_table[parent_ptr, len] is defined){
     return canon_table[parent_ptr, len];
  }
  else{
    // incrementally define canon for (parent_ptr, len)
    canon_table[parent_ptr, len] = unalloc_address;
    unalloc_address += len;
    return canon_table[parent_ptr, len];
  }
}
```

**Fig. 3.** The implementation of the canon function

An object $p$ of length $l$ in the state consists of the following bytes $\{x_p, x_{p+1}, \ldots, x_{p+l-1}\}$. Thus, its partial hash value is the following partial sum.

$$\mathcal{H}(p) = \sum_{i=p}^{p+l-1} r_i x_i \;\; mod\ m$$

The hash value of the entire state is the sum (modulo $m$) of the partial hash valuse of all objects in the state. The hash update function is simply

$$\mathcal{H}(S(p, p')) = \mathcal{H}(S) - \mathcal{H}(p) + \mathcal{H}(p') \;\; mod\ m$$