Uncomputation

MAISA KIRISAME*, University of Utah, USA
PAVEL PANCHEKHA*, University of Utah, USA

Program execution need memory. Program may run out of memory for multiple reasons: big dataset, exploding intermediate state, the machine have less memory then others, etc. When this happens, the program either get killed, or the operating system swaps, significantly degrading the performance. We propose a technique, uncomputation, that allow the program to continue running gracefully even after breaching the memory limit, without significant performance degradation. Uncomputation work by turning computed values back into thunk, and upon re-requesting the thunk, computing and storing them back. A naive implementation of uncomputation will face multiple problems. Among them, the most crucial and the most challenging one is that of breadcrumb. After a value is uncomputed, it's memory can be released but some memory, breadcrumb, is needed, so we can recompute the value back. The smaller a value is, the breadcrumb it leave will take up a larger portion of memory, until the size of a value is less then or equal to that of the size of breadcrumb, so uncomputing save no memory. maybe remove? The more value uncomputed, breadcrumb accumulate, until most of the memory is used to record breadcrumb. This prohibit asymptotic improvement commonly seen in memory-cosntrainted algorithms. We present a runtime system, implemented as a library, that is absolved of the above breadcrumb problem, seemingly storing recompute information in 0-bits and violating information theory.

Additional Key Words and Phrases: Do, Not, Us, This, Code, Put, the, Correct, Terms, for, Your, Paper

ACM Reference Format:
Maisa Kirisame and Pavel Panchekha. 2018. Uncomputation. *J. ACM* 37, 4, Article 111 (August 2018), 11 pages. https://doi.org/XXXXXXX.XXXXXXX

0.1 Motivation(1.5pg)

1 INTRO

Program consume memory. Program might use too much memory:

- The input might be huge. Editing a file require loading the whole file into memory; Autocomplete load and parse the whole codebase; Image editors will keep multiple replica of similar images.
- Exploding intermediate state: Model checker and Chess bot search through a huge state space, Compiler and Interprocedural analysis expand the AST, Deep Learning/Backpropagation create and keep intermediate value. Additionally, interactive/incrmental application such as Juypter Notebook, Undo/Redo Stack, Time travelling debugger also keep intermediate state around to revert back to an older state when needed.
- Low memory limit. wasm have a memory limit of 2GB, embedded device or poor people may have device with lower ram. A personal computer might be running multiple programs,

Authors' addresses: Maisa Kirisame, marisa@cs.utah.edu, University of Utah, P.O. Box 1212, Salt Lake City, Utah, USA, 43017-6221; Pavel Panchekha, , University of Utah, P.O. Box 1212, Salt Lake City, Utah, USA, 43017-6221.

0004-5411/2018/8-ART111 $15.00
https://doi.org/XXXXXXX.XXXXXXX

which must contend for a fixed memory limit. GPU and other accelerator use their own separate memory and usually is of smaller capacity then main memory.

Even when there is enough memory, it is still beneficial to use less memory. Operating System use memory for file cache, the program may have software cache, garbage collector fire less often give more free space, and extra space could be used to increase load or to execute other process.

One common-bought up solution to the memory problem is - to just buy more memory. However this only work to certain extents. Accelerators and wasm have a fixed memory limit and memory-limited algorithm can improve space consumption asymptotically, a feat that merely hardware improvement cannot catch up to. should i talk about banker paradox here?

However, the only generic technique for reducing memory consumption, swapping, come with great drawback. Swapping move a chunk of data from main memory onto disk, and upon re-requesting it, move the data from disk back into main memory. It degrade performance significantly as disk is far slower then CPU and main memory, and it is especially bad for functional programs as most values is functional programs are boxed, they typically have less cache locality, exacerbated by swapping which operate on pages granularity.

Facing the lack of a generic solution, developers resort to ad-hoc, case-by-case solution, such as cache size tuning, reducing heap size, creating algorithm that swap and uncompute. Such solution complicate the system, and as the developer is likely not an expert on memory-constrained solution, suboptimal design/implementation is often made.

We propose uncomputation, a technique to reduce memory consumption for arbitrary purely functional programs. When needed, uncomputation select some values in the object graph according to cache policy, reverting it back into thunk, releasing memory until it is needed again, in which case it will recompute said values and replacing the thunk with the values. We provided a monadic API behind uncomputation, such that mere mechnical translation is needed to apply our work. The monadic API also grant compatibility, saving developers headache from learning and converting between a full-fledge language.

1.1 Pebble game

The idea of uncomputation is not new. More specifically, uncomputation had been studied rather heavily by the TCS community, by modelling it as a 'pebble game'.

Pebble game is played on a computation graph, a directed acyclic graph where vertex represent data and edge represent computational dependency between data. The game is played with a fixed amount of identical pebble, representing memory, with each vertex possibly having one pebble on it.

It is played with the following rules:

- If a vertex have no pebble, but all it's parent have pebble, a pebble can be placed on it. This represent computing a value based on it's dependency, and stroing the result.
- A pebble can be removed from a vertex. This represent deallocating a value.
- The game is finished if a pebble is placed on a special vertex, the result.

The aim of the game is to finish it in the minimal amount of compute step.

The pebble game is a prolific area of TCS research, with many variation of pebble game, modeling swapping or partially reversible computation, or with different cost/constraint. Special casing on the computation graph, where it form a planar graph or a tree, had likewised be studied. However, in the general case, on the classical pebble game is very diffcult.

It is hard to use less pebbles. John Hopcrof prove that for a pebble game of size X, one can use X / log(X) amount of pebble, but in general, no less. recheck this no less claim. Moreover given a

computation graph, the minimal amount of pebble (alongside the strategy) is not only NP-complete, but also inapproximable.

give an example of a pebble game.

Our main focus is orthogonal to that of pebble games'. While the pebble game focus on finding good strategy, we focus on finding strategy effectively, using little space and time. In particular, under the lens of pebble game, the breadcrumb problem thus become 'how to store the computational graph without storing it', a seemingly impossible task.

1.2 Guarantee(0.5pg)

There are multiple guarantees our approach give.

Correctness guarantee: uncomputing will not effect the final result (preservation), and will not be in an infinite loop (progress). Essentially, uncomputing is transparent: the semantic is that of the same as if none happens at all.

Asymptotic guarantee: O(1) access time for all in-memory objects, and O(1) space for each in-memory objects. In particular, this imply that we are not using any space for uncomputed objects.

2 UNCOMPUTATION(5PG)

2.1 Motivation

2.2 Strawman

2.2.1 API. Below we give a monadic api for zombie. We will describe how to implement it later. figure out name for: the API, the naive implementation, the actual implementation Zombie provide a monadic api. This mean that we exposed a type constructor, Zombie: * -> *, with the following 3 operations:

- return: X -> Zombie X, turning a value into Zombie. Note that any top-level return (that is, not in a bind), is not uncomputable, and will always live in memory, as there is no methods of uncomputing it.
- bindN: Zombie X... -> (X... ~> Zombie Y) -> Zombie Y, a modified bind. While it accept only static-function that does not capture any variable, it bind multiple Zombie at once. This is because a closure will capture non-zombie values, so memory cannot be returned. Imagine the following function: M A -> M B -> M (A * B). An implementation via bind and return will capture A, causing said A to be non-evictable. note that bindN is of the same power as a closure allowing monadic bind, and one can convert between the two form via defunctionalization and iterated binding. We could potentially allow zombie-capture, but doing that complicate our semantic without adding power, and we could allow all capture, putting performance responsibility on the users, but that make reasoning about performance much harder.
- get: Zombie X -> X, returning the value. It only for use as an 'escape hatch', or to obtain the final result: using it destroy dependency information, causing worst space-time tradeoff.

Zombie externally speaking, behave as the Identity Monad - return and get is the identity, and bindN is the reversed apply. however, zombie will uncompute value behind the scene, recomputing them when needed, to save memory.

2.2.2 Point-Applicative Implementation. Below we give a evolutionary approach, starting from an implementation of a point-applicative API, which is strictly weaker but easier to implement and explain.

Zombie X = Input X | Calculated (Ref (Maybe X)) Exists T..., (Zombie T)..., (T... -> X)

A Zombie is an Input, in which case there is nothing we can do about it. Or it might be a calculated value from an ApN. We can then create a slot to store Maybe the result, alongside the thunk that compute it, with the dependency.

When the value is forced, we either return the input, or the Maybe X, if it is a Just, or we have to recalculate it. That is achieved by forcing the T's from the Zombie, then feeding it into the function, then storing and returning the result.

While the above implementation make sense, it suffer a fatal flaw: it is not a monad. This cause Uncomputation to be only usable for program without recursiveness, as such program will require join.

2.2.3 Monadic Implementation. Luckily such problem can be circumvented.

Zombie X = Input X | Calculated (Ref (Maybe X)) Replayer

Replayer = Exists T..., (Zombie T)..., (T... -> ()), Ref (List (Exists x, Zombie x))

Compared to the pointed-applicative version, there are multiple changes.

The biggest one is the change from a function with the dependency, to a single common type, the Replayer. The change stem from the fact that a single Bind may invoke multiple or no return, so a common Replayer type allow reuse from said multiple returned zombie.

Replayer, then, have to distinguish and fill value of those zombies. It does so by turning the function from a purely-functional calculator, to a imperative writer, looking for the Zombie and filling the slot. The zombies are taken from the reference list which store all Return. The list is a reference as this is form a cyclic dependency. But it's usage is very 'functional': Before full construction, it is cons-only, and once fully constructed, it will not be written anymore.

Get return X if it is an Input, or if it is an Calculated currently holding a value, otherwise it invoke the function, call get recursively on it, storing and returning the result.

The implementation have a global data structure, a stack of Replayer. upon entering bind, the Replayer stack grow temporarily, storing the dependency and the function modified. upon return is called, we will do a lookup to find the last context, using it to fill Zombie's Replayer field. If no such context is found, it is top level and thus an Input.

Get might be invoked by the user or by bind to turn Zombie X into X. Get on Input or Get on a Calculated with non-empty slot return said Value. Get on a Calculated with an empty-slot, however, need to recompute, by calling the replayer. This step recursively call get on Zombie T... input, invoke the function, which read the zombie list, reversing it, keeping a temporary local copy, and for each return, fill the intermediate result and pop the value off the list.

2.3 Guarantee

2.4 Problems

While the high-level idea seems straightforward, there are multiple subtle questions:

2.4.1 recursiveness. Recursiveness. A value, for example, a list, might be recursive. Furthermore, there might be sharing inside such datastructure. In such a case, how do we measure the size of a value, which is a valuable property guiding what to uncompute? Such value can be computed via recursively traversing all the descendant once, which is computationally expensive. There might also be external reference to part of said data structure, in which case the measured size is a poor estimate of what will be freed. Furthermore, not only cant we get size, staleness is also diffcult: likewise, the staleness information require traversing down the data structure, looking at every cell.

2.4.2 doppelganger. Partialness. Suppose a list is generated via anamorphism. We might want to uncompute the head of the list, but keeping some intermediate node inside the list. When

recomputing said list, we need to be able to retrieve such intermediate node, to avoid spending extra time to recompute them, and extra memory to store them twice.

2.4.3 uncomputation. Uncomputation candidates. Which value should we pick to uncompute?

2.4.4 breadcrumb. Breadcrumb. After a value is uncomputed, we need to store information needed to recompute said value. This become a bottleneck when most value is uncomputed, or when each value is small.

3 PLANT VS ZOMBIE(8PG)

3.1 Insight

3.2 Key Idea(1pg)

There are 3 key insights that enable the above guarantee:

- A global clock increasing on return/bind invocation. This not only give a way to quickly detect same objects for hashconsing, but as opposed to classical hash consing, the key itself have semantic meaning: < denote happens-before.
- Recomputing A also recompute what is inside A. This mean we can forget about some inner thunk, as long as we can redirect the pointers to such thunk to an outer thunk. One possibility is to track backpointers and fix them, but this take significantly more time and space. Instead, Track function exit time too. Entering and exiting form a range, and two separate range either form a nested relationship or disjoint relationship. We can then design a datastructure, such that lookup failure return a match one-level above.

this is old tex. merge.

There are multiple insights that allow us to solve the above 4 problems. Explaining them aid understanding the implementation of Zombie.

0: treat each cell as independent, ignoring all member relationship. this trivialize size finding.

Zombie allow recursive object, e.g. a list, a tree, or a graph. For those object, a part of the object can be evicted, and sharing does not affect size counting. This feature is implemented by ignoring recursiveness. All nesting need to loop through Zombie, and when it does that, the management of subsequent nested Zombie is completely handoff to it. Note that this allow keeping some value of the list in memory, even when there is previous node evicted!

1: hashconsing. to avoid the same expression producing multiple values, we introduce a type Idx, such that the same trace will be tagged with the same Idx. Zombie X hold Idx instead of X, and the actual values are stored in a datastructure. When return x try to create a Zombie X of tag idx, if idx is stored in the datastructure, that value is used instead, even though the computed value is equivalent to said value. This cause x to be immediately unreachable and can be released from memory. This allow evicting most of the list, only keeping some intermediate value. nowhere reachable via the object graph.

2: note that bind can be nested: the function in bind may call more bind. When such nesting occur, executing the higher-level bind will also re-invoke the lower level bind. Hence - it is not necessary to store the lower-level thunk (even though doing so improve performance, as it will execute less code).

With this in mind, if we can create a data structure imitating the bind/return execution of the program, we can remove non top-level nodes, so that lookup will return a higher-level result then the remove node, we solved the breadcrumb problem.

3.3 Implementation

Combining the 3 insights above we now sketch an implementation of Zombie. There is a global tock, which value begin from 0, increasing on every invocation of bind/return. A Zombie<X> hold an Integer, tock, instead of value of type X. The value is stored in the tock tree. Whenever a call to return or bind is finished, we put a Node onto the tock tree. The range of said node is the tock value at the begin and end of the invocation.

For return, the node contain the value.

For bind, the node contain the list of input Zombie, the output Zombie, and the function pointer.

In bind, we have to force values from Zombie X to X. This is done via looking it up from the tock tree. If the look up node is not a Value, but the Thunk, we need to rewind the tock to the beginning of said thunk, replay the value, get the value from the tock tree (todo: avoid infinite loop here), and restore the tock.

4 TOCK TREE API

we need a data structure, such that when after removing a node, looking said node up does not return Nothing, but rather return a less precise answer.

Below is an API summing up what we need.

Tock = u64

Range = Tock * Tock

Make: TockTree X

Insert: TockTree X -> Range * X -> ()

Per two call to Insert, their range must either be nested or non-overlapping.

Lookup: TockTree X -> Tock -> Option (Range * X)

Remove: TockTree X -> Range -> ()

geometric series

4.1 Tock Tree

Such data structure is implemented by a tree, where the tree structures mimic that of bind/return nesting structure. In particular, each node in the tock tree X have: a range children, stored in an avl-tree, ordered by their start-tock. The range of the children do not overlap. X

The top level node is special in that it does not have range nor children. Or, put in another way, we have a Tock-forest.

Given an AVL of node ordered by their start-tock, we can test if a tock X is in one of the node's range by finding the largest node smaller or equal to X, then seeing if said node include X. using this operation, we can implement all operations on tock-tree in a straightforward manner.

The key of the datastructure is the Lookup Function, which recursively traverse down the tree, until it found a node such that it's children does not the tock. In such case we return the last range and the value. This setup allow removing node, without LookUp returning None, as it will return the value higher-up instead. As the higher up function semantically imply, and will recreate all lower up function, removing will not destroy any information (in the information theoretic sense).

4.2 Recomputation

upon a get, one might find a Recomputer instead of a Value on the tock tree. When that happend, we push the tock of get to a 'request stack', then replay the function.

said replaying begin by rebinding (which might cause further replay), then executing the function. since we already know the exact tock we want, when encountering bind which cannot possibly include such tock (the function is already on the tock tree, and it's range does not include X), we

are free to skip such function, returning the stored return zombie instead. upon replaying the exact tock, we replace the tock on the request stack, with the value (crucially, not the zombie) return is invoked with.

note how the normal evaluation is eager as we have to assign tock to all function, but replay evaluation is lazy as we are free to skip once the metadata is present. This is a sad consequence of relying on tock.

4.3 formalization

4.3.1 language.

$$E := BaseTerm|Var|Abs(Var:T)...E|AppEE...|BindE...E|ReturnE \qquad T := BaseType|ZT|T... > T$$

STLC is weak, but I want to say that it is 'the same' under another type system. or better yet, despite the monadic structure type is inessential to our approach

4.3.2 typing rule.

$$\frac{}{Env| - BaseTerm : BaseType} \qquad \frac{v : TinEnv}{Env| - v : T} \qquad \frac{Var : T...| - E : T}{Env| - Abs(Var : T)...E : T... > T}$$

$$\frac{Env| - E : T...- > T \qquad (Env| - E : T)...}{Env| - AppEE... : T}$$

$$\frac{(Env| - E : T)... \qquad Env| - E : (E : T)... > T}{BindE...E : T} \qquad \frac{Env| - E : T}{Env| - ReturnE : ZT}$$

note that in the Abs rule, and the Bind rule, we throw away the old environment. this is because we do not allow closure. One could still implement another type representing closure or defunctionalized closure, different from the > type bindN need.

4.3.3 small step operational semantic.

$$V ::= BaseTerm|Abs...E|ZN \qquad \frac{v = VinEnv}{Env, H, v >> Env, H, V} \qquad \frac{Env, H, E >> Env', H', E'}{Env, H, AppEE... >> Env, AppE'E...}$$

$$\frac{Env, H, E. >> Env, H', E.'}{Env, H, AppV(V...)E.(E...) >> Env, H', AppV(V...)E.'(E...)}$$

$$\frac{}{Env, H, App((Var : T)...E)V... >> (Var = V)..., H, E}$$

$$\frac{Env, H, E. >> Env', H', E.'}{Env, Bind(V...)E.(E...)E >> Env, H', Bind(V...)E'(E...)E}$$

$$\frac{Env, H, E >> Env', H', E'}{Env, BindV...E >> Env, H', BindV...E'} \qquad Env, BindZV...(Var : T)...E >> (Var : V)..., E$$

$$\frac{Env, H, E >> Env', H', E'}{Env, H, ReturnE >> Env, H', E'} \qquad \frac{}{Env, H, ReturnV >> Env, H(x := V), Zx}$$

write about tock tree one weird thing about it is that our 'small step semantic' may actually take multiple step. This allow us to isolate out small step semantic, simplifying our correctness proofs. If wished, one could define some classical small step semantic, and prove bisimulation between the two.

another thing is that Env modification does not carry. I think this is normal. check.

also - a bit unsettling that Abs Evaluate the Function before the Argument, but Bind does it the other way. Should we unify the two, or make Bind look more like app?

4.4 Correctness Gurantee

4.4.1 *Progress.*

4.4.2 *Preservation.*

4.5 Asymptotic

4.6 not violating information theory

clearly, it is impossible to store information in 0-bits, so where and how much storage are we using to store the breadcrumbs?

as the program execute, the global tock counter will continuously increase, until it reach the 64-bit limit and overflow.

assume there is x live objects, and y dead objects, word size needed to store tock per live object is O(log(x + y)), so total memory size is O(x * log(x + y)) taking the derivative of the expression w.r.t. y, give O(x/(x + y)). this give a sub-constant result: as there are more and more breadcrumb, the cost of storing each of them decrease, approaching 0!

in practice, this number will not overflow. even in some extreme case where it does, making it into 128-bit will guarantee non-overflowness by being longer then the age of the universe. Even at 128-bit, the word size is still a small portion of the total overhead.

suprsingly, this 'reducing n space program into log n space program' is also seen in the famous TREEVERSE. it is unclear if this is a mere conincidence, or if there is any deeper connection.

4.7 OLD

All Zombies are threaded through a logical clock, a tock, stored as a u64. During execution, the clock increases whenever makeZombie and bindZombie are called.

A Zombie contains, theoretically speaking, contains only the tock that the Zombie is created at. To access the actual value of the Zombie, a ZombieNode, we rely on a global data structure. It will be explained later. (We also store a weakpointer to ZombieNode as a cache, to quicken access. This is just a cache, and is irrelevant to the rest of Zombie). A ZombieNode holds X but is inherited from the Object class, which contain a virtual destructor and nothing else. This essentially allow one to type erase ZombieNode of different type, to put them into the same data structure.

We implement Zombie via two global data structures, a log and a pool. The pool track all currently in-memory Zombie representation - it manage space The log record actions taken used for recomputation - it manage time

The pool holds a vector of uniqueptr<Phantom>, the class that manage eviction. It has api to evict objects, and api to score the profitability of eviction. When we are low on memory, we can find a value in this vector, call evict(), and remove it. This will free up memory in the log.

For every call to makeZombie and bindZombie, we keep track of when the function is called vs when the function exited. This establishes a tock range. For makeZombie, since it does not call any more functions, its interval length is 1, containing exactly the tock of the zombie. Tock range overlap iff one contain another, iff one function calls another function. We then store tock range, paired with

information on said function. For makeZombie we store a shared pointer to the ZombieNode. For bindZombie, we record the function used to compute it, as std::function<void(const std::vector<const void*>)>, alongside the tocks of all input Zombies, and the tock of output Zombie. We call it a Rematerializer. The input argument is type erased to void*, as Zombie is type polymorphic, but we need a type uniform interface here.

The log is some kind of interval tree. Each node in the tree stores its begin and end interval, alongside a value, and a BST from tock to sub nodes. The key of BST is the start of the sub node's interval. When we query the log via a tock, we will find a value with the closest interval containing said tock. This mean, we might query for a Zombie, but instead of getting a ZombieNode, we get a Rematerializer. In such a case, we replay the Rematerializer by fetching the ZombieNode from the tock (replaying recursively if necessary), setting the clock to begin of the function temporarily, and entering said function. Afterward we set the clock back, and fetch the value from a tree. When we set the clock via rematerialization, we always set it to a smaller value, thus guaranteeing termination.

The log itself, when replaying, also serve as a memo table, indexed by tock. When we want to create a Zombie, we will skip when such Zombie already exist in the log. When we want to run BindZombie, we return the result tock when the precise Rematerializer exist in the log. This allow us to not compute the same function multiple time, if we know its result, and allow us to not store the same Zombie multiple time.

Note that all non-top-levels node in the tree can be removed. When we do so, we will free up memory, and when we need the value again, we can always replay a node higher up.

This design allow us to store n log n amount of metadata per n alive(nonevicted) objects.

Thus, when a Rematerializer dominate no value, and another value dominate Rematerializer, we remove it. (todo: implement). Idea: manage this using pool.

There is a slight error with the above design: maybe the value is recomputed and put on the tree, but then evicted, and fetch will not return a ZombieNode but a Rematerializer. To fix this we introduce a global data structure called Tardis, consist of a tock and a shared ptr<ZombieNode>*. When we create Zombie, when the tock match that of tardis, it will write to the shared ptr, holding the value longer.

4.8 replaying semantic

Program evaluate differently in replaying mode then in non-replay mode, as relevant Node might already be on the tock tree. When

Every entering tock (lhs of a range) uniquely identify a return/bind.

5 CACHE POLICY(2PG)

note that the cache policy does not effect the correctness of the implementation, only the performance.

GDSF

5.1 Measuring compute time

measuring time is more complex then it seems.

need to be recursive and uncount recursive time.

5.2 recursive cache policy

5.3 union find

6 EVAL(3PG)

6.1 Small Program

List Program
- Pascal Triangle(2d, 3d)
- Interpreter for LC
- PE for LC
- Interpreter for IMP
- Compiler from IMP to LC
- String Serialization and Deserialization
- String Compression and Uncompression
- Parsing and Unparsing
- String = radix-tree/flat-array/linked-list
- KD-tree, on shortest distance and on ray tracing
- statistic ask
- Graph - DFS BFS MCTS
- where does undo redo fit in?

6.2 Composite Program

compose the programs to form larger program

7 CASE STUDY(2PG)

8 RELATED WORK(2PG)

8.1 Memory-Constrained Algorithm

Memory Limit is a common problem. Lots of different subfields of CS had experienced such problem, and developed specialized algorithm that trade space for time, by recomputing, for their own need. cite. In particular, a famous algorithm, TREEVERSE, had been re-discovered independently multiple times in unrelated subfields.

Keeping uncomputation at each subfield not only cause multiple re-discovery and re-implementation, wasting valuable researcher/programmer time, but also come short in integration. Suppose a complex, memory-hungry software have 2 sub-parts, both of which are memory hungry. How much uncompute responsibility should each part take? What happens if two part depend on each other?

8.2 Pebbling

we had talked about pebbling in intro. delete? Space time tradeoff is a known-topic in Theoretical Computer Science. It is mostly modeled as a "pebble game" quickly explain. It had been known that this problem is NP-complete and inapproximable. There are also generic result in this space. Hopcroft prove that for any program that take N time, it could be modified to take O(N / log(N)) space. However, the proof is non-constructive, and thus does not help building an automatic runtime like ours.

Our work mostly focus on the overhead problem - how to minimize the overhead of recording metadata needed for recomputation. How to make an API that is generic yet embeddable. In another words - our work is orthgonal to advances in pebble game. In fact, one could imagine replacing our cache policy with some of those algorithms.

We also provide a greedy cache policy that is both fast and make reasonably good decision on the list of benchmark, both real and synthetic, provided below, since the theoretical result is weak and non-constructive. There are some specialized result of pebble game on planar graph and on tree. One could imagine upon detecting planar graph/tree, switching from our cache policy to the theoretically optimal ones.

8.3 Memoization

Uncomputation is the dual to Memoization.

8.4 Garbage Collection

Garbage Collection is known to ocassionally suffer memory leak, when a live object hold reference to object no-longer used, as GC work by tracing pointers. There are some work that try to use liveness analysis to solve this problem, by detecting when a object is no longer used yet still pointed-to, then removing it. Needless to say, such approach will always be limited by rice theorem and there will always be garbage it cannot collect.

Zombie, when seen as a GC technique, offer an interesting approach to this problem. While garbage collector ask for permission to remove an object, instead zombie ask for forgiveness - as a value live longer and longer untouched, it will be more and more profitable for the cache policy to uncompute it, and once uncomputed, if it is dead object, it will not be used again, hence we sucessifully deallocate the memory. If our guess is wrong, however, we pay the price of recomputing when needed again. This distinction share resemblence to optimistic lock vs pessmistic lock.

8.5 Compression

Uncomputation is, fundamentally speaking, a form of compression.

Pigeon hole principle mean it is fundamentally impossible to have a generic compression algorithm. Uncomputation exploit the inductive bias that a program pointer is smaller then its memory consumption is larger.

9 MAYBE?

9.1 Constant Optimization

9.2 Tail Call

9.3 Non Tail Call

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009