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 than 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. 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

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, program slicing, 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.

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.

1.3 Insight

While the above asymptotic guarantee seems very strong, it is actually achievable via:

- Record undelimited context instead of closures. Undelimited context, when run long enough, will produce everything later.
- Assigning a monotonically increasing number, tock, to all zombie creation, and to all call/return/tailcall. This give each zombie a global unique id, so repeat creation of zombie do not create duplicate objects. It also give an ordering so we can find earlier context to replay.
- A data structure, tock tree. The tock tree is a binary search tree, that, on lookup failure, instead of failing (via exception/nullptr/optional), will return the closest match: the value associated with the largest key less then/equal to the input key. This design allow removal of arbitrary context, except the earliest one, as removing an earlier context merely cause lookup to return a further context, requiring more replay.

pavel look here I hate how we talk bad about a naive approach later, but explain our core idea here. I think we should not have naive approach at all, or have it before core idea. Also - 1.3 and 1.4 is new. What other thing do I need for the intro?

1.4 Contribution

Our contribution is as follow:

- A monadic API, with a naive implementation and why it doesnt work
- A implementation without problem above
- Formulation and proof of important properties

2 UNCOMPUTATION(5PG)

As our construction contain multiple parts, and might seems unintuitive to an outsider, we adopted an incremental approach to explain our work.

2.1 Point-Applicative

2.1.1 API. We start from an point-applicative api:

- pure: X -> Zombie X, turning a value into Zombie. Since this function works on raw value, there are no compute dependency recorded, and we can not uncompute any zombie created

this way.

- apN: (X... ~> Y) -> Zombie X... -> Zombie Y, a modified Ap.

  The function argument must be a static function, hence we denote it with a ~> arrow. This restriction is because as a library instead of a whole-language approach, closure might capture non-zombie argument. Storing such closure will cause memory managed by Zombie to extend it's lifetime, nullifying uncomputation on those Zombie.

  The above restriction lose expressivity. For example, it is no longer possible to write the function Zombie A -> Zombie B -> Zombie (A * B). To accommodate for the expressivity lost, Ap now become a polyvardic function, taking an arbitrary amount if Zombie, alongside a function of matching type.

  The two interface is equally expressive, via defunctionalization and via currying.

  We could potentially allow zombie-capture in the function, 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: in particular, using get, calling function f then pure instead of ApN will not register f in Zombie, so it is impossible to uncompute.

Note that an implementation of Zombie, aside from satisfying the above type, and classical law on applicatives, should furthermore behave semantically equivalently as a Identity type. That is, Zombie should not temper with the value stored inside in anyway. Of course, a efficient implementation of Zombie should be more then the Identity type, uncomputing and recomputing values in the user's back.

2.1.2 Implementation. The above API could be implement with this definition of Zombie:
   Zombie X = Input X | Calculated (Ref (Maybe X)) Exists T..., (Zombie T)..., (T... ~> X)

A Zombie is either 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 calling get on Zombie T, to get the T values, feeding it into the function, then storing and returning the result. This recompute process is recursive: a recompute might find its dependency missing, so further recompute is done on those. Furthermore, this recursive process also consume and might run out of memory, and when that happen Zombie need to uncompute during said recompute. Extra care must be given to avoid uncomputing what just got recomputed, as to not enter an infinite loop.

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 Monadic

2.2.1 API. The API is a slight twist from the above, only that apN (now bindN) return Zombie instead of raw Value.

- return: X -> Zombie X. We must relax the constraint from pure: any non-top-level call of return will not be uncomputable. This is because the monadic interface rely heavily on return.

- bindN: (X... ~> Zombie Y) -> Zombie X... -> Zombie Y. Like apN, it is polyvardic and does not allow closure. We flipped the argument order, putting the function before the monad, to emphasize symmetry with ap and with function application.

- get: Zombie X -> X. It is exactly the same as get from the applicative API.

2.2.2 Implementation. The change from Applicative to Monadic API pose several challenges.

- return now behave differently, base on whether it is top-level or not. Furthermore return need to find the arguments to bind. To do this, we need to introduce a global data structure, a stack of bind-context, for return to match on.
- bind may invoke zero or multiple return, as opposed to ap always 'returning' once. So when Zombie replay, it need to fill multiple said Zombie, instead of only one.

Given the above challenges, the Zombie type is now:

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. This allow multiple return in the same bind to share one replayer.

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 work on 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

do we want to proof the same thing three time? (one for the applicative version, one for the monadic version, and one for the final version)

2.4 Problems

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

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. pavel look here start reading here

3   PLANT VS ZOMBIE(8PG)

urgggh how do i transit from the naive implementation to this

3.1   SplayList

the name splaylist technically exist but not really. can we call it that? to begin explaining the core data structure, we have to explain a splay list. a splay list is a splay tree, fused with a doubly linked list. the doubly linked list maintain the ordering on all keys. template<typename K, typename V>
struct SplayList {
struct Node {
K k;
V v;
Node* parent;
Node* child;

Node* splayparent;
Node* splayleftchild;
Node* splayrightchild;
};
};
   huh. splaylist save one lookup. look like a normal splay tree will do. maybe we can move this into optimization
   A splaylist provide the following api:
get: SplayList K V -> K -> option<V>
put: SplayList K V -> K -> V -> ()
remove: SplayList K V -> K -> V -> ()
   with additionally,
get_key: SplayList K V -> K -> option<K>
has_precise: SplayList K V -> K -> bool
get_approx: SplayList K V -> K -> option<V>
remove_approx: SplayList K V -> K -> V -> ()
   The API could be classified into 3 groups:

- get/put/remove: this is the normal splay tree operation
- get/remove_approx: get/remove like above, but find the largest node smaller/equal to the current node, and operating on it, instead of failing.
- has_precise: whether the key exist, and get_key, the largest key smaller/equal to the input.

the approx version is implemented by descending to the leaf node just like a normal BST, but of the node have a greater key, visiting it's parent node via the doubly linked list pointers, exactly once.

3.2 Tock Tree

The core data structure in zombie, the tock tree, is a splaylist of tock to context. A context record all information needed to replay at that tock, alongside all zombie's internal content(ZombieNode) produced at that context.

Formally, a context at tock t contain the following: A function pointer we actually use std function (more costly) for better ergonomic A list of tock representing the input of said function pointer A Timespan recording how long the execution take A vector of shared ptr, storing ZombieNode of tock (t+1), (t+2)... (t+n)

With Tock Tree defined, a Zombie is merely a Tock paired with a weak pointer to zombie node, acting as cache for fast access.

3.3 Recording Semantic

While recording, we have a PartialContext garbage name. A PartialContext is a Context with the timespan replaced by a start time (as it is still running and recording), and with the vector of zombie node being filled. It also additionally keep a tock denoting when it started.

A PartialContext is created whenever bindzombie is tail-called. it record the current time, the current tock, and the function pointer alongside the argument to bindzombie. The old PartialContext is then finished (by subtracting the time recording, turning the instant into a duration) then putted onto the splaylist.

When zombie is created, the zombienode is put into the vector of the PartialContext.

3.3.1 supporting non-tail call. bindzombie may appear in more then tail call position. When this happend, upon reentering, a special context, where instead of the function pointers alongside input, the tock of the original context is stored. to support this PartialContext form a stack. tailcall/zombie modify the head of the stack, and on a non-tailcall bindzombie, the PartialContext is put onto the stack.

3.3.2 supporting top-level zombie. There is also a special, unreplayable context, consisted of only vector of zombie node, and a tock pointing to the last replayable-place, a top-level Context.

Note that under CPS transformation, there are no top-level zombie, and all call are tail call. However we still support the above two case for completeness reason.

3.4 Replaying Semantic

Replaying require a additional stack, the replay stack. The replay stack is a stack of tock paired with a zombienode(shared pointer of any). Replaying work just like normal playing, but on zombie creation it read the top of the stack, and if the tock match, modify the shared pointer to point to the zombienode.

Replaying is entered once we bind a value, but the value does not exist (as it had been evicted). Replaying can cause further replaying, and eviction might also happen inside replaying which will cause such further replaying.

3.5   proof sketch

progress: the replay stack move forward

preservation: under some conditionfigure out, on any given splaylist/replay stack/context stack, the evaluation is a bisimulation of normal evaluation semantic.

pavel I know this is not a triangle, but I think the triangle step is captured in the 'different splaylist condition'. pavel look here stop reading here

3.6   Insight

The key insight that solve the breadcrumb problem is to notice that bind are nested.

The function in a bind might call more bind, which themselves might call more bind, and so on. In fact, in a monadic translation of lambda calculus program, there is only one top-level bind, with all other bind nested inside it. This mean that in-principle, we only need to store this top-level function, freeing us from storing the rest.

To do this, we need to detach Replayer from Zombie, storing them instead on a global data structure. Zombie then store the index to said datastructure instead of storing Replayer directly. With a carefully designed data structure, explained later, we can then remove intermediate Replayer, as long as the index lookup will return some Replayer that represent a bind that will invoke the original bind, instead of failing. Bind now need to find and refill the Zombie object that is still in-memory. This can be accomplished by having Zombie X not storing X directly, but also stored on such data structure. A Zombie then, is nothing but merely an index to the data structure.

A simple, fast and slim index is logical time. As we also use physical time for other purpose, to avoid confusion, we will name our logical time tock. There is a global system clock, tock, that start from 0, increasing everytime bind or return is called, before actually executing them this is a bit redundant, but i want to stress. The tock at the beginning and at the end of bind/return execution will give us range, and bind/return nest if one ranger include another. Note that there are no non-nesting intercept: either a range include another, or two range have no common tock at all. This suggest our data structure should be a tree, and including range be represented by ancestor-children relationship. A lookup then find the smallest matching range, and return the data store on such node. An uncomputed value is then represented by an index that, when lookup, return a Replayer instead of the value.

We can then set the tock temporarily to the start of the range, re-run the function, then reset the tock back. When rerunning the function, if the same return node is encountered, the new value can be dropped directly, resolving us from the doppleganger problem. why not drop the old value? I think dropping the new is better performant-wise. maybe test. If the same bind node is encountered, the bind can be skipped, returning the final result (which is now stored in the Replayer rename). To avoid uncomputing result needed, there is an extra stack storing reference of optioanl values, paired with the tock. When replaying return, if the tock match, the value need to be written to the reference.

With the above design, looking back, by fixing breadcrumb, we solved doppleganger on the way, and recursiveness likewise become a non-problem: the lifetime of a zombie object is orthogonal to the lifetime of what it hold, so there is no longer any recursiveness: each node of a recursive object is managed by it's own individual zombie only.

3.7   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.

3.8 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. fix all bind. they now take the function before the arguments

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

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.

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

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

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

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

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

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

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 CALL STACK

While the above construction can

5.1 Tail Call

Call stack - get a node with inf right value onto tock tree, fix when 'actual' return

Clock stack - put the value on the node

C++ stack - trampoline

Replay stack - bounded by the tock tree size * intermediate node count, so it is linear w.r.t. tock tree.

5.2 Non Tail Call

By doing a continuation passing style transform all call become tail call. The continuation might grow in size but we can zombified it.

6 CACHE POLICY(2PG)

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

GDSF

6.1 Measuring compute time

measuring time is more complex then it seems.

need to be recursive and uncount recursive time.

6.2 recursive cache policy

6.3 union find

pavel look here old text from recursive cache paper. i think it is somewhat readable. problem: calculating recursive cost is expensive.

one may be tempted to solve this problem via dynamic programming: storing a recursive compute cost for each node in the graph, calculated by summing up it's own compute cost with recursive compute cost of that of it's parent.

such approach have two drawback: 0 - when upstream node got evicted/rematerialize, downstream node need to update their recursive cost, thus this is merely trading graph traversal at pull to graph traversal at push. 1 - will double count at diamond dependency, leading to potentially exponentially larger estimation.

adopting the approach of dtr should we say this?, we use a union find data structure.

We assign each node two evicted node is in the same eclass if they are both evicted, and one is the parent of another. additionally, transitivity equate node that is not immediate parent/children. note that this include more node then recursive children/parent. draw a graph to illustrate.

each eclass additionally store a compute cost. everytime a node enter a eclass via eviction, we add said node's compute cost to the eclass. when a node leave the eclass via rematerialization, we do the according subtraction.

this allow us to estimate compute cost in approximately O(1) time.

7 EVAL(3PG)

7.1 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
    FFT
    Graph - DFS BFS MCTS
    where does undo redo fit in?

7.2 Composite Program

compose the programs to form larger program

8 CASE STUDY(2PG)

9 RELATED WORK(2PG)

9.1 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?

Zombie not only provide an automatic method for memory-constraint algorithm, it also solve the constitutionality problem: by using a single central cache policy, responsibility is implicitly calculated and spreaded to the two subcomponent. Furthermore, by using a recursive cache policy, the union find, the dependency between the system is also taken into consideration.

9.2 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.

9.3 Memoization/Caching

Uncomputation is the dual to Memoization. While memoization and caching use extra space to store computational result to avoid recomputing, we are unstoring result and recomputing them to save memory.

9.4 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 sucessfully 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.

9.5 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 the memory consumption objects represent.

cooperative caching - opposite of us

membalancer - balancing multiple stuff, space time tradeoff

swapping? - slow demonstrated via gimp. cache + swap is interesting. cite some paper that manage both.

berger's oopsla paper - manage both cache and gc

FIFO Queues are All You Need for Cache Eviction

The CacheLib Caching Engine: Design and Experiences at Scale

Kangaroo: Caching Billions of Tiny Objects on Flash

A large scale analysis of hundreds of in-memory cache clusters at Twitter

Improving WWW Proxies Performance with Greedy-Dual-Size-Frequency Caching Policy

GreedyDual Web caching algorithm: exploiting the two sources of temporal locality in Web request streams

Replacement Policies for a Proxy Cache - introduce LRV

Johnny Cache: the End of DRAM Cache Conflicts (in Tiered Main Memory Systems)

Software-Defined Caching: Managing Caches in Multi-Tenant Data Centers

Caching less for better performance: Balancing cache size and update cost of flash memory cache in hybrid storage systems

My cache or yours? Making storage more exclusive

Cooperative Caching: Using Remote Client Memory to Improve File System Performance

FaaST: A Transparent Auto-Scaling Cache for Serverless Applications

OFC: an opportunistic caching system for FaaS platforms

FairRide: Near-Optimal, Fair Cache Sharing

GD-Wheel: a cost-aware replacement policy for key-value stores

Cliffhanger: Scaling Performance Cliffs in Web Memory Caches

DistCache: Provable Load Balancing for Large-Scale Storage Systems with Distributed Caching

An Analysis of Facebook Photo Caching

10   CONSTANT OPTIMIZATION