Uncomputation

MARISA KIRISAME*, University of Utah, USA
HENG ZHONG, Fudan University, P. R. China
TIEZHI WANG, Tongji University, P. R. China
SIHAO CHEN, Hangzhou Dianzi University, P. R. China
PAVEL PANCHEKHA*, University of Utah, USA

Programs need memory to execute, and when there isn't enough memory available, programs can swap, crash, or even get killed by the operating system. To fix this, we propose Zombie, a purely-function programming language implementation that allows programs to continue running (albeit much slower) while using much less memory than necessary to store all of the program's heap-accessible values. Zombie works by replacing pointers with abstract "tocks", each of which refers to a value via the point in time when it was allocated. This allows Zombie to evict computed values from memory and recompute them as necessary by rewinding and replaying program execution. Zombie's runtime stores a limited amount of metadata, much smaller than the total number of program steps, allowing it to shrink live memory dramatically. Thanks to a combination of optimizations and a powerful cache heuristic, Zombie is able to achieve memory reductions of up to XXX× while incurring slowdowns of XXX×.

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

ACM Reference Format:
Marisa Kirisame, Heng Zhong, Tiezhi Wang, Sihao Chen, and Pavel Panchekha. 2018. Uncomputation. *J. ACM* 37, 4, Article 111 (August 2018), 17 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRO

1.1 Motivation

Program execution consumes both space and time. While there is tons of research focused on reducing the time spent running a program, memory usage reduction has been consistently under-appreciated.

Yet, saving memory is still an important topic worth studying:

- Multi-tenancy. An end-user laptop executes multiple programs concurrently. As an example, a typical programmer might open an IDE to edit and compile code, Zoom for remote meetings, and additionally a browser with dozens of tabs open to look up information on the internet. Among them, the worst is the web browser, as each tab is its own separate process, with a renderer, a rule engine, and a JavaScript runtime. All software above consume a decent amount of RAM and contend between themselves for memory.

Authors' addresses: Marisa Kirisame, marisa@cs.utah.edu, University of Utah, P.O. Box 1212, Salt Lake City, Utah, USA, 43017-6221; Heng Zhong, hzhong21@m.fudan.edu.cn, Fudan University, 220 Handan Road, Yangpu District, Shanghai, P. R. China, 200437; Tiezhi Wang, 2152591@tongji.edu.cn, Tongji University, 4800 Caoan Road, Jiading District, Shanghai, P. R. China, 201804; Sihao Chen, 21201112@hdu.edu.cn, Hangzhou Dianzi University, No. 2 Street, Qiantang District, Hangzhou, Zhejiang, P. R. China, 310018; 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

- Huge input. While a typical image might have a resolution of around 1000 x 1000, large images might reach a size 100 to 1000 times larger than that. Images editors and viewers typically thrash or oom upon processing images of such size. Similarly, text editors typically process text of < 1MB, and fail on logs or other large text files reaching GBs. Likewise, IDEs will parse and analyze the source code of a project and then store the analyzed results for auto-completion. Upon working on a huge project such IDEs will crash.
- Intermediate state. A modern computer can read and write memory at a speed of GB per second. Without memory reclamation techniques such as garbage collection, memory will run out in a matter of seconds. However, some applications are essentially out of reach from garbage collection and must keep most if not all intermediate states around. These applications include time traveling debugger, Jupyter Notebook, reverse mode automatic differentiation for deep learning, incremental computation/hash consing, and algorithms that use search, such as model checker and chess bot.
- Low memory limit. Wasm has a memory limit of 2GB, and embedded devices or poor people may have devices with lower RAM. GPU and other accelerator use their own separate memory and usually are of smaller capacity than main memory.

A typical and generic solution to reduce memory consumption is by uncomputation. Uncomputation trades space for time, by tagging values with the metadata that created it(thunk), and when memory is low, deallocate values that might be used later on, recomputing them back with the thunk when the value is needed again.

Another generic solution is swapping, whereupon low memory, pages are swapped to disk and swapped back when needed again. While swapping had been implemented for basically all operating systems, it is an especially bad solution for functional and object-oriented languages, as those languages allocate and deallocate lots of small objects, yet swapping work on the granularity of pages, unable to separate out the hot objects, that had been recently accessed from the code objects, that had not been accessed.

Typically, uncomputation is implemented on an ad-hoc, case-by-case basis. When software consumes more memory than desired, its programmer can (1) look at the program, (2) decide which part is using excessive memory, then (3) add custom data type to record the thunk, alongside the code to replace it (4) implement a cache eviction policy to decide what data to evict(uncompute). Another solution is to analyze the algorithm carefully, replacing the algorithm with a specialized version with recomputation in mind, see gradient checkpointing/iterative deepening depth-first search/island algorithm.

Needless to say, this process is extremely cumbersome, taking precious developers' time away from more critical tasks such as optimizing CPU usage or implementing new features, so a generic, automatic approach is needed to reduce memory consumption.

1.2 Failed Attempt

At first glance, uncomputation is easy, as its cousin, lazy evaluation, is heavily studied. In lazy computation, an object might be represented as thunks, which, when needed, is then computed, and the thunk is replaced by the value. A solution to uncomputation is to modify thunk, so that, upon computing, the old function is still kept. This allows the thunk to uncompute as if it had never been recomputed.

```
Lazy Evaluation: type 'a lazy = MkLazy of ('a, unit -> 'a) either ref

Uncomputation: type 'a uncompute = MkUncompute of 'a option ref * (unit -> 'a)
```

Just like lazy evaluation, we can then uniformly lift all values on top of our modified value type, inserting code that forces weak head normal form upon data access and has some caching policy to decide what to uncompute, which merely rewrites the reference back into the empty optional value.

However, there are multiple problems with this approach:

(1) size measurement. We want to gather statistics to decide what values to evict, and one of the most important statistics is the memory an object consumes. However, the heap forms a complex object graph, and fast cardinality estimation is highly complex.
(2) dopple ganger. The translation will lift a list into nested uncompute. Now suppose variable X holds a list, while variable Y holds a sublist of X, sharing the same representationbad wording do not know how to fix. When X is uncomputed and recomputed, the corresponding Y part will be duplicated, so there will be two representations of Y in memory. In the extreme case, such duplication can force the program to take exponentially more memory.
(3) bread crumb. A thunk captures other values as a free variable, which contains a thunk part, capturing more value. This recursive process captures all values that the current value transitively computes on. Since we still need memory to store the thunk itself, and since a thunk is typically larger than an object, anything we gained on uncomputation, will be offset by the metadata.

While the above three problems seem difficult, we can observe they are all but mere manifestations of recursiveness. In particular, size measurement and dopple ganger deal with recursive objects and breadcrumb deals with recursive thunk. This indicates that a solution that handles recursiveness will naturally solve the three problems at once.

1.3 A recursive Solution

To combat recursiveness we take heavy inspiration from Abstracting Abstract Machine (AAM). AAM lifts abstract interpretation onto programming languages with arbitrary features. The critical problem there is likewise recursiveness, as a naive lifting might allow the lattice infinite merge, causing the analysis to non-terminate. AAM proposes to use an abstract machine that abstracts over pointers, allocations, and lookups to manage recursiveness.

Likewise, the three recursiveness problems listed above can be handled similarly. We started from a purely functional language, specifying its semantics with the CEK machine. We can then abstract over pointers/allocations/lookups similarly.

Most importantly, our key insight is that by careful handling of our core data structure, the tock tree, which will be explained later, we can remove a value, alongside the metadata(thunk) on that value, yet still recompute it later. This solves the breadcrumb problem which renders the naive approach unsalvageable.

Alongside our solution is proof that it is both correct and efficient. Our correctness proof states that uncomputing will not affect the final result(preservation), and will not infinite loop(progress). The result will be the same as what gets computed by the raw, uncomputation-free semantic, no matter what we evicted. Our efficiency proof states that if our memory consumption is O(n), where n is the amount of live objects: objects that take O(1) time to force into weak head normal form. In particular, this implies asymptotically we use no space to store any evicted object, yet we somehow can access the metadata that recomputes them!

We implemented our solution, alongside an eviction policy that is both fast to compute and makes better decisions over classical eviction policies like LRU, random, and GDSF.

Name	x	::=	A set N of distinct names
Expression	C	::=	x | Let $x = A$ in B | $\lambda x,\ A$ | $F(A)$ | (A, B) | $A.0$ | $B.1$ | Left A | Right B | Case A of { Left $x \Rightarrow L$ | Right $y \Rightarrow R$ }

Fig. 1. The syntax of λ_Z. Lower case letters represent variables and upper-case letters represent expressions. The semantics of each expression is standard.

Environment	E	::=	$[(N, V)]$
Value	V	::=	Clos $E\ N\ C$ | VProd $V\ V$ | VLeft V | VRight V
Continuation	K	::=	Done | KLookup K | KLet $N\ E\ C\ K$ | $\text{KApp}_0\ E\ C\ K$ | $\text{KApp}_1\ E\ N\ C\ K$ | $\text{KProd}_0\ E\ C\ K$ | $\text{KProd}_1\ V\ K$ | KZro K | KFst K | KLeft K | KRight K | KCase $E\ N\ C\ N\ C\ K$
State		::=	Eval $E\ E\ K$ | Apply $K\ V$

Fig. 2. Definitions of values, environments, and states for the CEK machine. Note that E represents environments, and C represents expressions. Kontinuations K represent computation steps that can be applied to a value and are named after the corresponding expression type.

2 OVERVIEW

The tock tree serves as a cache insert this sentence somewhere

Section: The CEKR Machine (???) Replay Stack the section with lots of Greeks argue about progress

Section: Implementation (3pg long) Heuristic Loop Unrolling

Key question: How to get the replay stack small? Key question: Garbage Collection/Eager Eviction

Summarize the meeting into a key step

Double O(1)

3 CEK MACHINES AND HEAPS

Zombie works on a purely functional language with products, sum types, and first-class functions called λ_Z. For simplicity, we treat the language as untyped in this paper. The syntax of λ_Z is shown in Figure 1; its semantics are standard. Additional features supported by the full Zombie implementation, such as primitive types and input/output, are not relevant to recomputation and are described in **??**.

Importantly, λ_Z is purely functional, meaning programs are totally deterministic: evaluating a given expression in a given environment always returns the same result. This is essential for Zombie to work correctly.

3.1 CEK Machine

The key insight of Zombie is to assign a unique identifier to any value ever allocated during the program's execution. Conceptually, it identifies each value with the execution step that allocated it. It does this using a variant of the CEK machine.

The CEK machine is a well-known abstract machine for executing untyped lambda calculi, where the machine state consists of three parts:

(1) Control, the expression currently being evaluated.
(2) Environment, a map from the free variables of the Control to their values.

(3) Kontinuation, which is to be invoked with the value the Control evaluates to.

In this work, it is convenient to split the evaluation phase from the invocation of the continuation, resulting in a variation of the CEK machine with the following machine state:

$$\mathsf{State} ::= \mathsf{Eval}\ C\ E\ K \mid \mathsf{Apply}\ K\ V$$

The Eval state stores the classic control-environment-kontinuation, while Apply stores a kontinuation and the value to which it is being applied. The precise syntax of values and kontinuations are given in Figure 2; their semantics is standard.

Execution in the CEK machine involves a series of transitions between these machine states; in other words, it is a transition system. To run a program C in the CEK machine, one creates the initial state (Eval C {} Done), with its Control set to the expression to evaluate, Environment empty, and Kontinuation set to a special Done kontinuation. The rules of the CEK machine are then used to transition from one machine state to another, until finally reaching the state Apply Done V. In that state, V is the result of evaluating C.[1] The full CEK transition relation, for our variant of the CEK machine, is given in **??**.

For example, to evaluate a function application $F(X)$ and apply the kontinuation K to the result, the CEK machine first evaluates the function argument:

$$\frac{}{\mathsf{Eval}(F(X), E, K) \rightsquigarrow \mathsf{Eval}(F, E, \mathsf{KApp}_0\ K\ X)}$$

Note that the argument X is stored in the KApp_1 kontinuation, to be evaluated at a later point. The KApp_0 kontinuation basically represents the one-hole context $\lambda v, v(X)$.

Eventually, assuming the program is well-typed, F will be evaluated to a closure value (Clos), at which point the CEK machine will transition to an Apply state, which will then transition to evaluating the argument:

$$\frac{}{\mathsf{Apply}(\mathsf{KApp}_0\ E\ X\ K, \mathsf{Clos}\ E'\ N\ C) \rightsquigarrow \mathsf{Eval}(X, E, \mathsf{KApp}_1\ E'\ N\ C\ K)}$$

Note that the components E', N, and C of the closure are now stored in the KApp_1 kontinuation, which basically represents the one-hole context $\lambda v, (\mathsf{Clos}\ E'\ N\ C)(v)$.

Finally, when the argument X has been evaluated to an argument, the closure is called:

$$\frac{}{\mathsf{Apply}(\mathsf{KApp}_1\ E\ N\ C\ K, V) \rightsquigarrow \mathsf{Eval}(C, E[N = V], K)}$$

Note that the kontinuation K here refers to the original receiver of the value of $F(X)$, which has been carefully passed through a series of transition steps. In other words, during the execution of a program, the kontinuation K forms a kind of singly-linked list representing the stack of one-hole program contexts that evaluation is proceeding within.

A notable property of the CEK machine is that each transition performs a bounded amount of work. Contrast this to a traditional operation semantics, where the semantics of a Case statement might include a rule like:

$$\frac{\Gamma \vdash e \longrightarrow \Gamma' \vdash e'}{\Gamma \vdash \mathsf{Case}\ e\ \mathsf{of}\ \{\ \mathsf{Left}\ x \Rightarrow L \mid \mathsf{Right}\ y \Rightarrow R\ \} \longrightarrow \Gamma' \vdash \mathsf{Case}\ e'\ \mathsf{of}\ \{\ \mathsf{Left}\ x \Rightarrow L \mid \mathsf{Right}\ y \Rightarrow R\ \}}$$

In such a small-step semantics, derivations form a tree, and the antecedent of the inference rule might require an arbitrarily-large derivation, and thus an arbitrary amount of computation. In

[1] Note that, because λ_Z is untyped, it contains non-terminating programs using, for example, the Y combinator. In the CEK machine, these programs create an infinite sequence of machine states that do not include a terminating Apply Done V.

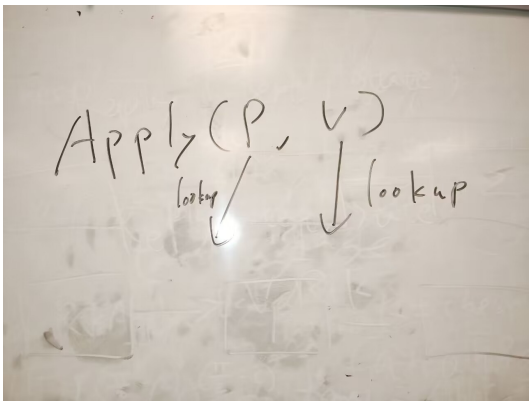

Fig. 3. Unlike a traditional small-step or big-step operational semantics, derivations (and thus computations) in the CEK machine form a flat list, with each step requiring a bounded amount of computation.

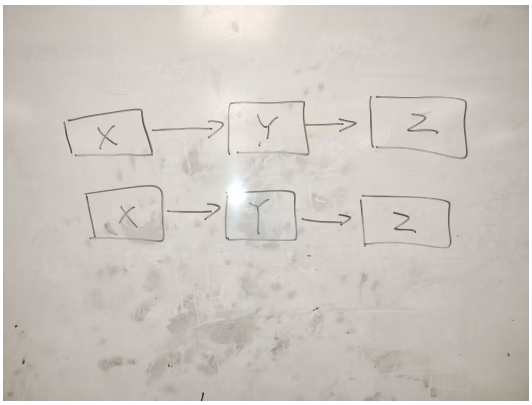

Fig. 4. Because the CEK machine is linear and deterministic, it can be "rewound" to a prior program point and be guaranteed to perform exactly the same sequence of states as during its prior execution.

the CEK machine, no such steps exist, and derivations in the CEK machine form a flat list. This property of the CEK machine is illustrated graphically in Figure 3.

Importantly, the CEK machine is linear and deterministic: every CEK machine state transitions to at most one state. That, in turn, means that if we were to "rewind" a CEK machine, putting it in an earlier state, it would transition through exactly the same sequence of states in the exactly same order. This determinism or "replayability" is essential for Zombie to work, and is illustrated graphically in Figure 4.

One key property guaranteed by determinism is that, for a given initial program C, the machine state at any point during C's execution can be uniquely identified by how many steps have been executed since the initial state. In other words, the initial state is identified with the number 0, the next state it transitions to with the number 1, and so on. Formally, this can be done by extending the state with a tock and extending every transition $S_1 \rightarrow S_2$ with an increment $t, S_1 \rightarrow t + 1, S_2$. This state number, which we call a "tock", is logically unbounded, but in our implementation, it is stored as a 64-bit integer. In our implementation, that suffices for several decades of runtime on current hardware.

Heap	H	::=	An abstract key value store
Pointer to X	$P\langle X\rangle$	::=	Key into heap with value type X
Lookup		:	$(P\langle X\rangle, H) \to X$
Alloc		:	$(X, H) \to (P\langle X\rangle, H)$

Fig. 5. The abstract heap / pointer interface for the CEK-H machine.

3.2 Heap Memory

Because we are interested in the memory usage of λ_Z program, we now extend the CEK machine with an explicit heap and explicit pointers. Values V are now implemented as pointers $P\langle\text{VCell}\rangle$ to "value cells", which can be closures, products, and sums and which in turn contain values, that is, pointers. Kontinuations K are likewise replaced with pointers $P\langle\text{KCell}\rangle$ to "continuation cells". The CEK machine state is now extended with a heap H; we call the extended machine CEK-H. The heap, pointers, and related methods are shown in Figure 5; the new definitions of values and kontinuations are shown in **??**.

CEK transitions now need to lookup or store values on the heap using the $\text{Lookup}(P, H)$ and $\text{Alloc}(X, H)$ functions. For example, evaluating a function application $F(X)$ now requires allocating a KApp_0 cell:

$$\frac{\text{Alloc}(\text{KApp}_0\ K\ X, H) = (P, H')}{\text{Eval}(F(X), E, K), H \rightsquigarrow \text{Eval}(F, P), H'}$$

Meanwhile, applying a KApp_0 kontinuation requires dereferencing the value pointer and then allocating the KApp_1 kontinuation:

$$\frac{\text{Lookup}(V, H) = \text{Clos}\ E'\ N\ C \qquad \text{Alloc}(\text{KApp}_1\ E'\ N\ C\ K, H) = (P', H')}{\text{Apply}(\text{KApp}_0\ E\ X\ K, V), H \rightsquigarrow \text{Eval}(X, E, P'), H'}$$

The full Eval and Apply transition relation is given in **??**.

Importantly, every CEK machine step (whether Eval or Apply) makes at most one call to Lookup and at most one call to Alloc. This means that each allocation the program makes can be uniquely identified by the tock for the machine state when it is allocated. This identification is the core abstraction that drives Zombie.

4 UNCOMPUTING AND RECOMPUTING

4.1 Tock

The key insight of Zombie is to introduce an abstraction layer between the executing program and the heap. This abstraction layer will allow the heap to transparently discard and recompute the program's intermediate values. To do so, we use the CEK machine to refer to each intermediate value by the index of the computation step that created it. Since each CEK step allocates at most one cell, this correspondence is injective. In other words, instead of storing pointers that refer to memory locations, we will store pointers, which we call "tocks", that refer to points in time—that is, CEK step indices.

In our runtime, these "tocks" are implemented as 64-bit integers, though in our model we will treat them as logically unbounded integers. There is a global "current tock" counter, which starts at 0 and is increased by 1 at every transition step in the abstract machine and every allocation. The cell allocated at step i is then referred to by the tock i, and that tock can then be used as a pointer, stored in data structures, looked up by later computations steps, and the like.

Note that, due to the determinism and linearity of the CEK machine, tocks are strictly ordered and the value computed at some tock i can only depend on values computed at earlier tocks. Moreover, we can recreate the value at tock i by merely rerunning the CEK machine from some earlier state to that point. Because the CEK machine is deterministic, this re-execution will produce the same exact value as the original. Because our pointers refer to abstract values, not to specific memory locations, the heap can thus re-compute a value as necessary instead of storing it in memory.

In other words, the way Zombie works is that the heap will store only some of the intermediate values. The ones that aren't stored will instead be recomputed as needed, and the heap will store earlier CEK states to facilitate that. As the program runs, intermediate values can be discarded from the heap to reduce its memory usage.

Because a value can be recomputed from *any* earlier CEK state, it will also turn out to be possible to store relatively few states. Therefore, overall memory usage—for the stored intermediate values and the stored states—can be kept low. In Zombie, we can store asymptotically fewer states than intermediate values, allowing us to asymptotically reduce memory usage.

4.2 Tock Tree

In Zombie, the heap is implemented by a runtime data structure called the tock tree. The tock tree maps tocks to the cells allocated by that step; in other words, the tock tree implements the mapping between tocks and their actual memory locations. Because values can be evicted to save memory, however, not all tocks have a mapping in the tock tree. Instead, every tock that *is* present in the tock tree will also store its state. To recompute an evicted value at some tock i, the tock tree will find the largest state at tock $j < i$ and replay from that tock to tock i.

Each node in the Tock Tree stores both a memory cell and a state. Specifically, consider the execution of the CEK machine from step t. In step $t + 1$ it allocates a cell; it then performs a computation in step $t + 2$. So the node at tock t contains both the cell allocated at step $t + 1$ as well as the state *afterward*, at step $t + 2$. A transition might not allocate any new cells, in such a case the Node only stores a state and no memory cell. To be more specific, if the node represents the execution of the CEK machine at step t, it will only contain the CEK state at step $t + 1$. Also note that a state, in this context, does not include every info needed for execution: in particular it does not include the tock tree, alongside any data that guide recomputation. More specifically a state denotes either a Eval $C\ E\ K\ Tock$ or a Apply $K\ V\ Tock$.

One thing worth noticing is that continuations are also reified. This means we treat them as like values, lifting pointers into tock, and storing continuations onto the tock tree. This allows us to also uncompute continuation as well, by evicting the node on the tock tree holding said continuation. This is crucial as traversing a deep data structure non-tail-recursively produces a continuation chain of equal depth. In contrast to a ordinary runtime, in which this is a slight but inevitable unfortunate, in our case, the continuation storage can become the bottleneck of the system.

In order to make this operation efficient, the tock tree is organized as a binary search tree. In a binary search tree, arbitrary keys (tocks) can be looked up in $O(\log(n))$ time, where n is the size of the tock tree, and when a key is not found, the largest key smaller than it can be easily found. Then the heap can replay execution from that point to compute the desired tock.

4.3 Replay Continuation

In order to replay for a cell at a tock t, we need to store (1) the tock t that we replay to, so we can stop once the value corresponding to t is found and (2) the original state before replaying, resume execution at that point, so we can stop once t is found, and the original state before replaying, so we can resume execution at that point. This is essentially a continuation, which we call the replay continuation (RK).

Value	=	~~P<VCell>~~	Tock
Continuation	=	~~P<KCell>~~	Tock
State	::=	~~Eval *C E K* | Apply *K V*~~	Eval *C E K Tock* | Apply *K V Tock*
Node	=		(Maybe(KCell | VCell), State)
Query	:	(TockTree, Tock)	-> (Tock, Node)
Insert	:	(TockTree, Tock, Node)	-> TockTree

Fig. 6. Tock Tree API. Note that we deliberately avoid dictating what node gets uncomputed, to decouple uncomputation/recomputation with selecting what to uncompute. Instead, a node might be dropped during insertion into TockTree. One might e.g. set a limit on the amount of nodes in the Tock Tree.

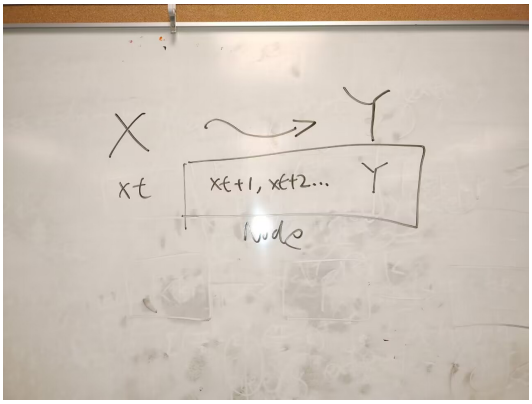

Fig. 7. a node in the tock tree

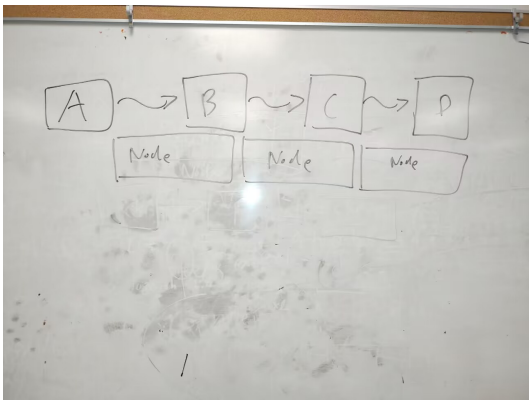

Fig. 8. the tock tree with multiple nodes

Note that it is different from the normal continuation in the following way:

(1) continuation is an expr with a hole of type expr, while replay continuation is a state with a hole of type cell
(2) there are fewer replay continuation types than continuation: this is because only the state that lookup a cell needs a replay continuation; a state that does not, does not need a replay,

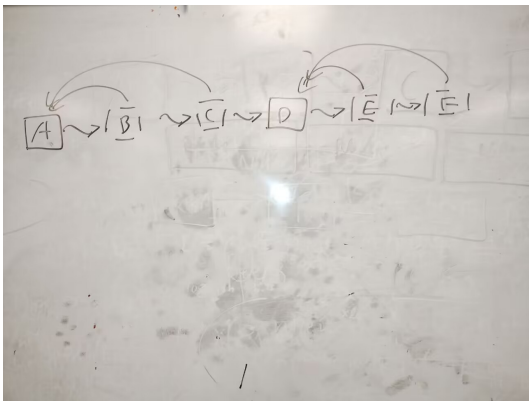

Fig. 9. lookup failure return the latest earlier node

ReplayContinuation	RK	::=	NoReplay | Replaying Tock RH RK |
ReplayHole	RH	::=	RHLookup V Tock |
			RHCase Env N C N C K Tock |
			RHZro K Tock | RKFst K Tock |
			RHApp Env E K Tock
Replay	R	::=	(State, RK)

and so does not need replay continuation. An example is Eval of a function application, as it only focuses on the function, pushing the arguments onto the continuation stack.

Note that, during the replaying process, more lookups might be issued, and those lookups might need more replay - replay is recursive. Just like the classical continuation at the CEK machine, the Replay Continuation needs to be recursive and form a stack as well.

4.4 CEKR Machine

With the idea of tock, tock tree, and the replay continuation, we can now define an abstract machine, the CEKR Machine, that can uncompute and replay values.

The CEKR Machine consists of 6 parts:

(1) Control, the expression currently being evaluated.
(2) Environment, a map from the free variables of the Control to their values.
(3) Kontinuation, which is to be invoked with the value the Control evaluates to.
(4) Replay, the replay continuation.
(5) The current tock
(6) The tock tree

In addition to the *Eval* and *Apply* modes in the CEK machine, there is a *Return* mode as well. This is an ephemeral mode, indicating that we have found a cell we are replaying for and will start playing from the top of the replay continuation at the next step.

$$\text{Machine} ::= (\text{State} \mid \text{Return Cell}), \text{rk}, \text{TockTree}$$

4.4.1 PostProcessing. The CEKR Machine executes by repeatedly computing the next node, and then applying a PostProcess step. the PostProcess step has two responsibilities: it inserts the node onto the tock tree, and checks if the node contains the cell the replay continuation is requesting. If

so, it means the current transition has found the cell we are replaying for, and our ending state should be the Return state with said cell. Otherwise, it transits to the original state.

$$\frac{}{CheckReturn : (t, node, rk) \longrightarrow State|Returncell} \qquad \frac{}{CheckReturn(t, (\text{Nothing}, st), rk) = st}$$

$$\frac{t + 1! = t'}{CheckReturn(t, (\_, st), \text{NoReplay}) = st}$$

$$\frac{}{CheckReturn(t, (\text{Just } cell, st), \text{Replaying}(t + 1)rhrk) = Return\ cell}$$

$$\frac{t + 1! = t'}{CheckReturn(t, (\text{Just } cell, st), \text{Replaying}t'rhrk) = st}$$

$$\frac{}{PostProcess : (t, cell, rk, TockTree) \longrightarrow Machine}$$

$$\frac{}{PostProcess(t, node, rk, tt) = CheckReturn(t, node, rk), rk, insert(tt, t, node)}$$

The transition is now split into 3 different cases:

(1) The happy path: in the CEKR machine, looking up from the heap is converted into querying the tock tree. The querying is successful, returning a node with a cell that matches the tock we are querying, and our transition can proceed as normal, producing a node, which is a pair of the cell produced and the state it transits to. PostProcessing is then applied.
(2) The sad path: when querying fails, returning a node that does not contain a matching cell, replay is needed. To replay a cell on tock t, we push the old state alongside t onto the replay continuation, and transit to the state in the node. Note that there is no PostProcessing in this case as a node is not created.
(3) The return path: when the machine enters the *Return* mode, it indicates that a query once failed, but we found the corresponding cell by replaying. By looking it up from the replay continuation, we can now re-execute the state that failed the query, as if the query result is a node containing said cell. No querying is needed in this case, as there is at most one lookup in each CEK transition, and the corresponding query had just been recomputed. Like the happy path, PostProcessing is needed to insert into the tock tree and to recursively return.

5 IMPLEMENTATION

We implemented the zombie runtime as a C++ library, alongside a C++ compiler that link to and call the library.

5.1 Tock Tree

To exploit the temporal/spatial locality, and the 20-80 law of data access (cite?), the tock tree is implemented as a slight modification of a splay tree.

This design grant frequently-accessed data faster access time. Crucially, consecutive insertion take amortized constant time.

The tock tree is then modified such that each node contain an additional parent and child pointer. The pointers form a list, which maintain an sorted representation of the tock tree. On a query, the tock tree do a binary search to find the innermost node, then follow the parent pointer if that node

$$\frac{st = \mathsf{Apply}(cell, v, (t+1)) \qquad node = (\mathrm{Nothing}, st)}{Return\ cell, \mathsf{Replaying}\ \_(RHLookup\ v\ t) rk, tt \rightsquigarrow PostProcess(t, node, rk, tt)}$$

$$\frac{st = \mathsf{Eval}(L, E[LN := X], K, (t+1)) \qquad node = (\mathrm{Nothing}, st)}{Return\ (\mathsf{VLeft} X), \mathsf{Replaying}\ \_(RHCase\ E\ LN\ L\ RN\ R\ K\ t), tt \rightsquigarrow PostProcess(t, node, rk, tt)}$$

$$\frac{st = \mathsf{Eval}(R, E[RN := Y], K, (t+1)) \qquad node = (\mathrm{Nothing}, st)}{Return\ (\mathsf{VRight} Y), \mathsf{Replaying}\ \_(RHCase\ E\ LN\ L\ RN\ R\ K\ t), tt \rightsquigarrow PostProcess(t, node, rk, tt)}$$

$$\frac{st = \mathsf{Apply}(K, X, (t+1)) \qquad node = (\mathrm{Nothing}, st)}{Return\ (\mathsf{VProd}\ X\ Y), \mathsf{Replaying}\ \_(RHZro\ K\ t), tt \rightsquigarrow PostProcess(t, node, rk, tt)}$$

$$\frac{st = \mathsf{Apply}(K, Y, (t+1)) \qquad node = (\mathrm{Nothing}, st)}{Return\ (\mathsf{VProd}\ X\ Y), \mathsf{Replaying}\ \_(RHFst\ K\ t), tt \rightsquigarrow PostProcess(t, node, rk, tt)}$$

$$\frac{st = \mathsf{Eval}(X, E\ , (t+1), (t+2)) \qquad node = (\mathrm{Just}(\mathsf{KApp1}E'NCK), st)}{Return\ (\mathsf{Clos}E'\ N\ C), \mathsf{Replaying}\ \_(RHApp\ E\ X\ K\ t), tt \rightsquigarrow PostProcess(t, node, rk, tt)}$$

Fig. 10. CEKR Transition: Returning

is greater then the key. This process is not recursive: the parent pointer is guaranteed to have a smaller node then the input key, as binary search will yield either the exact value, or the largest value less then the input, or the smallest value greater then the input.

5.2 Picking Uncomputation Candidate

5.3 Eviction

This allow us to remove any non-leftmost node from the tock tree. After the removal, the query that originally return the removed node, will return the node slightly earlier then that, which we can then replay to regenerate the removed node. In fact, this is the implementation of uncomputation in our system, and any non-leftmost node can be removed, to save memory at any given time. move to implementation section

Note that the guarantee we prove is independent of our policy that decide which value to uncompute (eviction policy).

5.3.1 Union Find.

5.3.2 The Policy.

5.3.3 GDSF.

5.4 Language Implementation

For implementation simplicity and interoperability with other programs, zombie is implemented as a C++ library, and the Cells are ref-counted. Our evaluation compiles the program from the applicative programming language formalized above(give name), to C++ code.

$$\frac{Query(tt, K) = (K-1, \text{Just} kcell) \qquad st = \text{Apply}(kcell, N, t+1) \qquad node = (\text{Nothing}, st)}{\text{Eval}(N, E, K, t), rk, tt \rightsquigarrow PostProcess(t, node, rk, tt)}$$

$$\frac{Query(tt, K) = (X, st) \qquad X! = K-1}{\text{Eval}(N, E, K, t), rk, tt \rightsquigarrow st, \text{Replaying } K(RHLookup\ N\ t)rk, tt}$$

$$\frac{st = \text{Eval}(X, E, t+1, t+2) \qquad node = (\text{Just KLet } A\ K\ C\ E, st)}{(\text{Eval}(\text{Let } A\ B\ C, E, K, t), rk), tt \rightsquigarrow PostProcess(t, node, rk, tt)}$$

$$\frac{st = \text{Eval}(X, E, t+1, t+2) \qquad node = (\text{Just Clos } E(\text{fv}) \cdots N\ C, st)}{(\text{Eval}(\text{Lam } N\ C, E, K, t), rk), tt \rightsquigarrow PostProcess(t, node, rk, tt)}$$

$$\frac{st = \text{Eval}(X, E, t+1, t+2) \qquad node = (\text{Just KApp}_0\ E\ X\ K, st)}{(\text{Eval}(\text{App } F\ X, E, K, t), rk), tt \rightsquigarrow PostProcess(t, node, rk, tt)}$$

$$\frac{st = \text{Eval}(X, E, t+1, t+2) \qquad node = (\text{Just KProd}_0\ K\ R, st)}{(\text{Eval}(\text{Prod } L\ R, E, K, t), rk), tt \rightsquigarrow PostProcess(t, node, rk, tt)}$$

$$\frac{st = \text{Eval}(X, E, t+1, t+2) \qquad node = (\text{Just KZro } K, st)}{(\text{Eval}(\text{Zro } X, E, K, t), rk), tt \rightsquigarrow PostProcess(t, node, rk, tt)}$$

$$\frac{st = \text{Eval}(X, E, t+1, t+2) \qquad node = (\text{Just KFst } K, st)}{(\text{Eval}(\text{Fst } X, E, K, t), rk), tt \rightsquigarrow PostProcess(t, node, rk, tt)}$$

$$\frac{st = \text{Eval}(X, E, t+1, t+2) \qquad node = (\text{Just KLeft } K, st)}{(\text{Eval}(\text{Left } X, E, K, t), rk), tt \rightsquigarrow PostProcess(t, node, rk, tt)}$$

$$\frac{st = \text{Eval}(X, E, t+1, t+2) \qquad node = (\text{Just KRight } K, st)}{(\text{Eval}(\text{Right } X, E, K, t), rk), tt \rightsquigarrow PostProcess(t, node, rk, tt)}$$

$$\frac{st = \text{Eval}(X, E, t+1, t+2) \qquad node = (\text{Just KCase } LN\ L\ RN\ R\ E, st)}{(Eval(\text{Case } X\ LN\ L\ RN\ R, E, K, t), rk), tt \rightsquigarrow PostProcess(t, node, rk, tt)}$$

Fig. 11. CEKR Transition: Eval

5.5 Optimization

5.5.1 Fast access path. Querying the tock tree for every value is slow, as it requires multiple pointer traversal. To combat this, each Value is a Tock paired with a weak reference, serving as a cache, to the Cell. When reading the value, if the weak reference is ok, the value is return immediately. Otherwise the default path is executed, and the weak reference is updated to point to the new Result.

$$\frac{Query(tt, K) = (X, (\_, st)) \qquad X! = K - 1}{\mathsf{Apply}(\mathsf{KLookup}K, V, t, rk), tt \rightsquigarrow st, \mathsf{Replaying}\ K\ (RHLookupVt)rk, tt}$$

$$\frac{Query(tt, K) = (K - 1, (\mathrm{Just}cell, \_)) \qquad st = \mathsf{Apply}(cell, V, t + 1) \qquad node = (\mathrm{Nothing}, st)}{\mathsf{Apply}(\mathsf{KLookup}K, V, t), rk, tt \rightsquigarrow PostProcess(t, node, rk, tt)}$$

$$\frac{st = \mathsf{Eval}(C, Env[A := V], K', t + 1) \qquad node = (Nothing, st)}{\mathsf{Apply}(\mathsf{KLet}\ A\ E\ C\ K', V, t), rk, tt \rightsquigarrow PostProcess(t, node, rk, tt)}$$

$$\frac{Query(tt, V) = (X', (\_, st)) \qquad X'! = V - 1}{(Apply(\mathrm{Just}\ KApp0EnvXK, V, t), rk), tt \rightsquigarrow st, \mathsf{Replaying}V\ (RHAppEnvXKt)rk, tt}$$

$$\frac{Query(tt, V) = (V - 1, (\mathrm{Clos}Env'NC), \_) \qquad st = \mathsf{Eval}(X, Env, t + 1, t + 2) \qquad node = (\mathrm{Just}KApp1E'NCK', s}{\mathsf{Apply}(\mathsf{KApp0}\ E\ X\ K', V, t), rk, tt \rightsquigarrow PostProcess(t, node, rk, tt)}$$

$$\frac{st = \mathsf{Eval}(C, Env[N := V], K', t + 1) \qquad node = (Nothing, st)}{\mathsf{Apply}(\mathsf{KApp1}EnvNCK', V, t), rk, tt \rightsquigarrow PostProcess(t, node, rk, tt)}$$

$$\frac{st = \mathsf{Eval}(R, Env, t + 1, t + 2) \qquad node = (\mathrm{Just}KProd1VEnvK, st)}{\mathsf{Apply}(KProd0EnvRK, V, t), rk, tt \rightsquigarrow PostProcess(t, node, rk, tt)}$$

$$\frac{st = \mathsf{Apply}(K, t + 1, t + 2) \qquad node = (\mathrm{JustVProd}LV, st)}{Apply(KProd1LK, V, t), rk, TT \rightsquigarrow PostProcess(t, node, rk, tt)}$$

$$\frac{Query(tt, V) = (X, (\_, st)) \qquad X! = V - 1}{Apply(KZroK, V, t), rk, tt \rightsquigarrow st, \mathsf{Replaying}\ V\ (RHZroKt)rk, tt}$$

$$\frac{Query(tt, V) = (V - 1, (VProdXY, \_)) \qquad st = \mathsf{Apply}(K, X, t + 1) \qquad node = (\mathrm{Nothing}, st)}{Apply(KZroK, V, t), rk, tt \rightsquigarrow PostProcess(t, node, rk, tt)}$$

$$\frac{Query(tt, V) = (X, (\_, st)) \qquad X! = V - 1}{\mathsf{Apply}(KFstK, V, t), rk, tt \rightsquigarrow st, \mathsf{Replaying}\ V\ (RHFstKt)rk, tt}$$

$$\frac{Query(tt, V) = (V - 1, (VProdXY, \_)) \qquad st = \mathsf{Apply}(K, Y, t + 1) \qquad node = (Nothing, st)}{\mathsf{Apply}(KFstK, V, t), rk, tt \rightsquigarrow PostProcess(t, node, rk, tt)}$$

Fig. 12. CEKR Transition: Apply

5.5.2 Loop Unrolling. To avoid frequent creation of node object, and their insertion to the tock tree, multiple state transition is packed into one.

$$\frac{st = \mathsf{Apply}(K, t+1, t+2) \qquad node = (\mathsf{Just}\ VLeftV, st)}{\mathsf{Apply}(KLeftK, V, t), rk, tt \rightsquigarrow PostProcess(t, node, rk, tt)}$$

$$\frac{st = \mathsf{Apply}(K, t+1, t+2) \qquad node = (\mathsf{Just}\ VRightV, st)}{\mathsf{Apply}(KRightK, V, t), rk, tt \rightsquigarrow PostProcess(t, node, rk, tt)}$$

$$\frac{Query(tt, V) = (X, (\_, st)) \qquad X! = V - 1}{\mathsf{Apply}(KCaseEnvLNLRNRK, V, t), rk, tt \rightsquigarrow st, \mathsf{Replaying}V(RHCaseEnvLNLRNRKt)rk), tt}$$

$$\frac{Query(tt, V) = (V-1, (\mathsf{Just}\ VLeftX, \_)) \qquad st = \mathsf{Eval}(L, E[LN := X], K', t+1) \qquad node = (Nothing, st)}{Apply(KCaseEnvLNLRNRK, V, t), rk, tt \rightsquigarrow PostProcess(t, node, rk, tt)}$$

$$\frac{Query(tt, V) = (V-1, (\mathsf{Just}\ VRightY, \_)) \qquad st = \mathsf{Eval}(R, E[RN := Y], K', t+1) \qquad node = (Nothing, st)}{Apply(KCaseEnvLNLRNRK, V, t), rk, tt \rightsquigarrow PostProcess(t, node, rk, tt)}$$

Fig. 13. CEKR Transition: Apply

5.6 Bit counting

6 FORMAL GUARANTEE

6.1 Correctness

the cek in this section does not use heap at all! in fact I dont think we really need the heap version. I think we should rename the cek with pointer into cekm, and explain it only for introductory purpose (or not at all).

6.1.1 Tock Tree Setup. In order to reason about the CEKR machine, we needed to reason about nodes in the tock tree, even if such node is evicted.

While we can reify the tock tree insertions into a list, forming into a trace semantic and reasoning on top of the list, it is easier to allow querying the tock tree for evicted data. In our formalization, we assume that the tock tree does not really evict; However, each node is now paired with a boolean bit, indicating whether it had been evicted or not. We will then have a function query_ghost : (TockTree, Tock) -> (Bool, Node) that return the Node, along with a boolean indicating whether the node is present.

6.1.2 CEK/CEKR connection. With query_ghost, we can connect value from the CEKR machine to the CEK machine. More specifically, we can now define a Sem(standing for semantic) function, that convert a CEKR-value to a CEK-value under a given tock tree. The function work by recursively calling query_ghost.

Like wise, we can define equality on continuation, and on state. Additionally, we define 'equality modulo eviction' between two Tock Tree, to be that the two Tock Tree contain the same node, but they might be under different eviction status. Formally speaking, forall Tock, the two Tock Tree must return two equal node, but the eviction bit might differ.

6.1.3 Ghost Stepping. Alongside query_ghost, is the idea of ghost_stepping this name suck. Just like the state transition on CEKR, ghost_stepping is also a transition on State, TockTree tuple. However, unlike the classical state transition, ghost_stepping use query_ghost in place of query,

which retrieve the exact node, even if the node is evicted. ghost_stepping is unimplementable, but just like our formalization of Tock Tree and query_ghost, it is purely for formalization purpose. As ghost_step does not need any replay, it does not have the replay continuation. ghost_step serve as a bridge between CEK and CEKR, connecting the two semantic. Once this is completed, we can only talk about correspondence between ghost_step and normal step, inside CEKR, without mentioning the CEK machine at all.

Lemma: CEK stepping is deterministic.

proof: trivial.

Lemma: Ghost stepping is deterministic.

proof: trivial. note that this require the tock tree being deterministic itself. write down this requirement in the right place.

Lemma: Ghost stepping is congruent under eqmodev.

proof: ghost stepping do not read from eviction status.

6.1.4 Well Foundness. A tock tree T is well-founded is this the right word? if:

(1) The left most node exist and is not be evicted. (root-keeping)
(2) Forall node N inserted at tock T, N only refer to tock < T. (tock-ordering)
(3) Forall node pair L R (R come right after L in the tock tree), if (L.state, T) ghost_step to (X, XT), X = R.state and T eqmodev XT. (replay-correct)
(4) Additionally, a (state, tock tree) pair is well-founded if state is in the tock tree. (state-recorded)

Lemma: ghost stepping preserve well-foundedness. proof:

(1) root-keeping is maintained by the tock tree implementation and is a requirement.
(2) tock-ordering is maintained - check each insert.
(3) replay-correct is maintained: the transit-from state must be on the tock tree due to state-recorded. If it is not the rightmost case, due to replay-correct we had repeatedly inserted a node. Since ghost-stepping is congruent under eqmodev replay-correct is maintained. If it is the rightmost state, a new node is inserted. All pair except the last pair is irrelevant to the insertion, because due to tock-ordering they cannot refer to it. For the last pair, ghost-step must reach the same state due to determinism in ghost-stepping.
(4) state-recorded is maintained: we just inserted said state.

6.1.5 Main Theorem. Theorem: under well-foundness, CEK-step and ghost-stepping preserve equivalence.

Formally: given CEK-step X, CEKR state Y, if (X, H) = (Y, T) and T is well-founded, (X, H) $\rightsquigarrow$ (X', H'), (Y, T) $\rightsquigarrow$ (Y', T'), (X', H') = (Y', T')

proof: wellfoundness ensure we do not write to old state and replace old node with other values. other part of proof is trivial.

The above theorem establish a connection between the CEK machine and the CEKR machine, by ignoring the replaying aspect of the CEKR machine, and only using it as if all values are stored in the tock tree (even though some nodes might already be evicted!). We will next connect ghost-stepping with regular stepping in the CEKR machine.

Lemma: stepping without changing rk is the same as ghost-stepping

proof: trivial

Lemma: if stepping add to rk, the moment it change back, is the same as single ghost-stepping

proof: due to well-foundness the cell RApply recieve must be the same as the same cell on the tock tree but inaccessible. Note that the stepping made here cannot add new node as it is bounded by the tock on the replay continuation, which is on the tock tree.

Lemma: it must change back

proof: by lexicalgraphical ordering on rk and on state

Theorem: X, rk, tt step-star Y, rk, tt' -> X tt ghost-step-star Y, tt", with tt' eqmodev tt'

proof: the step-star can be decomposed into sequence of no-rk-change step, or a push, with multiple step, and finally a pop. either case is covered by a lemma above.

6.2 Performance

6.2.1 Pebble Game. Let $G = (V, E)$ be a directed acyclic graph with maximum in-degree of k, where V is a set of vertices and E is a set of ordered pairs (v_1, v_2) of vertices.

For each $v \in V$, we can

- place a pebble on it if $\forall u \in V$ that $(u, v) \in E$, u has a pebble, which means all predecessors of v have pebbles.
- remove the pebble on it if v already has a pebble.

The goal is to place a pebble on a specific vertex and minimize the number of pebbles simultaneously on the graph during the steps.

6.2.2 Translation to pebble game.

6.2.3 Theorem. memory consumption is linear to amount of object with O(1) access cost

7 EVALUATION