Uncomputation

MARISA 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. Ironically, in a applicative language, due to boxing all values are small. This mean uncomputation, implemented naively, will only consume more memory, defeating the purpose. We present a runtime system, implemented as a library, that is absolved of the above breadcrumb problem.

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

ACM Reference Format:
Marisa Kirisame and Pavel Panchekha. 2018. Uncomputation. *J. ACM* 37, 4, Article 111 (August 2018), 16 pages.
https://doi.org/XXXXXXX.XXXXXXX

1 INTRO

2 OVERVIEW

The tock tree serve 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 key step

Double O(1)

3 CORE LANGUAGE

Zombie works on a purely functional language with products, sum types, and first-class functions called λ_Z. For simplicity in this paper, we treat the language as untyped. The syntax of λ_Z is shown in Figure 3; its semantics are standard. The full Zombie implementation supports additional features, such as primitive types and input/output; **??** describes how these features are layered on top of the core implementation described here.

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

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

Importantly, because Zombie is purely functional, programs are totally deterministic, in the sense that 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 an 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.

We should use C for expressions, to match the CEK terminology.

In our variation of the CEK machine, we split the evaluation phase from the invocation of the continuation, resulting in a machine state that looks like this:

$$\text{State} ::= \text{Eval } E \; Env \; K \mid \text{Apply } K \; V$$

In other words, our CEK machine can be in an Eval or an Apply state; the Eval stores the classic control, environment, and kontinuation, while the Apply state stores just a continuation and a value. The precise syntax of values and kontinuations are given in Figure 5.

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 first sets up an initial state (Eval C {} Done) whose Control is the expression to evaluate, whose Environment is empty, and whose Kontinuation is a special Done continuation. 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 of C.[1]

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 be something like:

$$\frac{\Gamma \vdash e \rightarrow^* \text{Left } \Gamma' \vdash V}{\Gamma \vdash \text{Case } e \; x \; e_l \; y \; e_r \rightarrow \Gamma', x : V \vdash e_l}$$

Here the antecedent of the inference rule might perform arbitrarily many steps, and thus an arbitrary amount of computation; derivations thus form a tree. In 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 1.

3.2 Determinism and Tocks

Importantly, the CEK machine is linear and deterministic. This means that 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 the exact same sequence of states in the exact same order. This determinism or "replayability" is essential for Zombie to work, and is illustrated graphically in Figure 2.

[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.

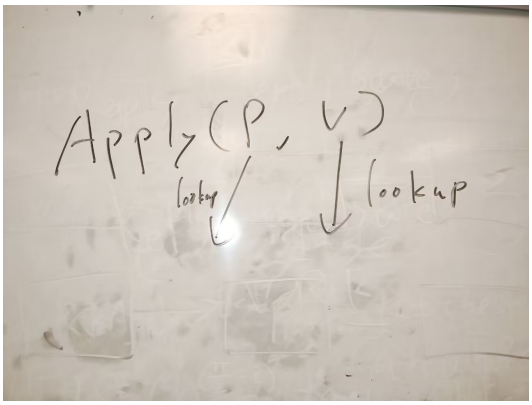

Fig. 1. the machine does a small constant amount of pointer lookup

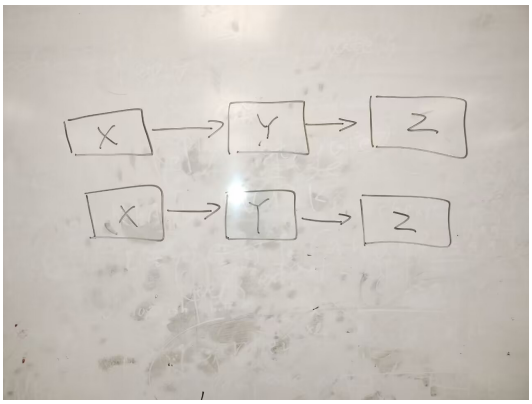

Fig. 2. the deterministic, linear nature of the CEK machine

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. 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.

3.3 Heap Memory

Because we are interested the total memory usage of λ_Z program, our variant of the CEK machine includes an explicit heap and explicit pointers. That is, in our formalization values V are formalized as pointers $P\langle \text{VCell} \rangle$ to "value cells". Values cells—which can be closures, products, and sums—in turn contain values, that is, pointers.

During execution, the CEK machine looks up these values using an explicit heap H. To interact with the heap, the CEK machine uses two functions. Lookup : $(P\langle X \rangle, H) \to X$ dereferences a pointer in the heap. Alloc : $(X, H) \to (P\langle X \rangle, H')$ stores a value in the heap, returning a pointer to it and the updated heap. Transitions in the CEK machine call Lookup and Alloc in order to make Eval and Apply steps, as shown in Figure 7 and Figure 8.

Name	N	::=	A set of distinct names
Expr	E	::=	N | Let x = A in B | $\backslash x.\ A$ | $f(x)$ | (x, y) | $x.0$ | $x.1$ | Left x | Right y | Case x of Left $a \Rightarrow L$ || Right $b \Rightarrow R$

Fig. 3. The syntax of λ_Z.

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

Fig. 4. Heap API

Continuation	K	::=	$P\langle$KCell$\rangle$
KCell		::=	Done | *KLookup K* | KLet *N Env E K* | KApp_0 *Env E K* | KApp_1 *Env N E K* | KProd_0 *Env E K* | KProd_1 *V K* | KZro *K* | KFst *K* | KLeft *K* | KRight *K* | KCase *Env N E N E K*
Value	V	::=	$P\langle$VCell$\rangle$
VCell		::=	Clos *Env N E* | VProd *V V* | VLeft *V* | VRight *V*
Environment	*Env*	::=	$[(N, V)]$
State		::=	Eval *E Env K* | Apply *KCell V*

Fig. 5. Definitions for the CEK Machine

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 where it is allocated. This identification is the core abstraction that drives Zombie's implementation.

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 at 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.

$$\frac{}{\mathsf{State} \rightsquigarrow \mathsf{State}} \qquad \frac{}{\mathsf{Eval}(N, Env, K) \rightsquigarrow \mathsf{Apply}(K, Env(N))}$$

$$\frac{}{\mathsf{Eval}(\mathsf{Let}\ A = B\ \mathsf{in}\ C, Env, K) \rightsquigarrow \mathsf{Eval}(B, Env, \mathsf{KLet}\ A\ K\ C\ Env)}$$

$$\frac{}{\mathsf{Apply}(\mathsf{KLet}\ A\ Env\ C\ K, V) \rightsquigarrow \mathsf{Eval}(C, Env[A := V], K)}$$

$$\frac{}{\mathsf{Eval}(\backslash N.E, Env, K) \rightsquigarrow \mathsf{Apply}(K, \mathsf{Clos}\ Env(\mathrm{fv}) \cdots N\ E)}$$

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

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

$$\frac{}{\mathsf{Apply}(\mathsf{KApp}_1\ Env\ N\ E\ K, V) \rightsquigarrow \mathsf{Eval}(E, Env[N := V], K)} \quad \frac{}{\mathsf{Eval}((L, R), Env, K) \rightsquigarrow \mathsf{Eval}(L, Env, \mathsf{KProd}_0\ K\ R)}$$

$$\frac{}{\mathsf{Apply}(\mathsf{KProd0}\ Env\ R\ K, V) \rightsquigarrow \mathsf{Eval}(R, Env, \mathsf{KProd}_1\ V\ K)}$$

$$\frac{}{\mathsf{Apply}(\mathsf{KProd}_1\ L\ K, V) \rightsquigarrow \mathsf{Apply}(K, \mathsf{VProd}\ L\ V)} \qquad \frac{}{\mathsf{Eval}(X.0, Env, K) \rightsquigarrow \mathsf{Eval}(X, Env, \mathsf{KZro}\ K)}$$

$$\frac{}{\mathsf{Apply}(\mathsf{KZro}\ K, \mathsf{VProd}\ X\ Y) \rightsquigarrow \mathsf{Apply}(K, X)} \qquad \frac{}{\mathsf{Eval}(X.1, Env, K) \rightsquigarrow \mathsf{Eval}(X, Env, \mathsf{KFst}\ K)}$$

$$\frac{}{\mathsf{Apply}(\mathsf{KFst}\ K, \mathsf{VProd}\ X\ Y) \rightsquigarrow \mathsf{Apply}(K, Y)} \qquad \frac{}{\mathsf{Eval}(\mathsf{Left}\ X, Env, K) \rightsquigarrow \mathsf{Eval}(X, Env, \mathsf{KLeft}\ K)}$$

$$\frac{}{\mathsf{Apply}(\mathsf{KLeft}\ K, V) \rightsquigarrow \mathsf{Apply}(K, \mathsf{VLeft}\ V)} \qquad \frac{}{\mathsf{Eval}(\mathsf{Right}\ X, Env, K) \rightsquigarrow \mathsf{Eval}(X, Env, \mathsf{KRight}\ K)}$$

$$\frac{}{\mathsf{Apply}(\mathsf{KRight}\ K, V) \rightsquigarrow \mathsf{Apply}(K, \mathsf{VRight}\ V)}$$

$$\frac{}{\mathsf{Eval}(\mathsf{Case}\ X\ \mathsf{of}\ \mathsf{Left}\ LN \Rightarrow L \mid\mid \mathsf{Right}\ RN \Rightarrow R, Env, K) \rightsquigarrow \mathsf{Eval}(X, Env, \mathsf{KCase}\ LN\ L\ RN\ R\ Env)}$$

$$\frac{}{\mathsf{Apply}(\mathsf{KCase}\ Env\ LN\ L\ RN\ R\ K, VLeftV) \rightsquigarrow \mathsf{Eval}(L, Env[LN := V], K)}$$

$$\frac{}{\mathsf{Apply}(\mathsf{KCase}\ Env\ LN\ L\ RN\ R\ K, VRightV) \rightsquigarrow \mathsf{Eval}(R, Env[RN := V], K)}$$

Fig. 6. Abstract Machine Transition: No Pointer

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

$$\frac{}{\mathsf{State}, H \rightsquigarrow \mathsf{State}, H} \qquad \frac{\mathrm{Lookup}(K, H) = KCell}{\mathsf{Eval}(N, Env, K), H \rightsquigarrow \mathsf{Apply}(KCell, Env(N)), H}$$

$$\frac{\mathrm{Alloc}(\mathsf{KLet}\ A\ K\ C\ Env, H) = (P, H')}{\mathsf{Eval}(\mathsf{Let}\ A = B\ \mathsf{in}\ C, Env, K), H \rightsquigarrow \mathsf{Eval}(B, Env, P), H'}$$

$$\frac{\mathrm{Lookup}(K, H) = KCell \qquad \mathrm{Alloc}(\mathsf{Clos}\ Env(\mathrm{fv}) \cdots N\ E, H) = (P, H')}{\mathsf{Eval}(\backslash N.E, Env, K), H \rightsquigarrow \mathsf{Apply}(KCell, P), H'}$$

$$\frac{\mathrm{Alloc}(\mathsf{KApp}_0\ K\ X, H) = (P, H')}{\mathsf{Eval}(F(X), Env, K), H \rightsquigarrow \mathsf{Eval}(F, P), H'} \qquad \frac{\mathrm{Alloc}(\mathsf{KProd}_0\ K\ R, H) = (P, H')}{\mathsf{Eval}((L, R), Env, K), H \rightsquigarrow \mathsf{Eval}(L, Env, P), H'}$$

$$\frac{\mathrm{Alloc}(\mathsf{KZro}\ K, H) = (P, H')}{\mathsf{Eval}(X.0, Env, K), H \rightsquigarrow \mathsf{Eval}(X, Env, P), H'} \qquad \frac{\mathrm{Alloc}(\mathsf{KFst}\ K, H) = (P, H')}{\mathsf{Eval}(X.1, Env, K), H \rightsquigarrow \mathsf{Eval}(X, Env, P), H'}$$

$$\frac{\mathrm{Alloc}(\mathsf{KLeft}\ K, H) = (P, H')}{\mathsf{Eval}(\mathsf{Left}\ X, Env, K), H \rightsquigarrow \mathsf{Eval}(X, Env, P), H'}$$

$$\frac{\mathrm{Alloc}(\mathsf{KRight}\ K, H) = (P, H')}{\mathsf{Eval}(\mathsf{Right}\ X, Env, K), H \rightsquigarrow \mathsf{Eval}(X, Env, P), H'}$$

$$\frac{\mathrm{Alloc}(\mathsf{KCase}\ LN\ L\ RN\ R\ Env, H) = (P, H')}{\mathsf{Eval}(\mathsf{Case}\ X\ \mathsf{of}\ \mathsf{Left}\ LN \Rightarrow L \mid\mid \mathsf{Right}\ RN \Rightarrow R, Env, K), H \rightsquigarrow \mathsf{Eval}(X, Env, P), H'}$$

Fig. 7. Abstract Machine Transition: Eval

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 machine states in order 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 machine state, it will also turn out to be possible to store relatively few machine states. Therefore, overall memory usage—for the stored intermediate values and the stored machine states—can be kept low. In Zombie, we can store asymptotically fewer machine 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

$$\frac{\text{Lookup}(K, H) = \mathit{KCell}}{\text{Apply}(\mathit{KLookup}\ K, V), H \rightsquigarrow \text{Apply}(\mathit{KCell}, V), H}$$

$$\frac{}{\text{Apply}(\text{KLet}\ A\ \mathit{Env}\ C\ K, V), H \rightsquigarrow \text{Eval}(C, \mathit{Env}[A := V], K), H'}$$

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

$$\frac{}{\text{Apply}(\text{KApp}_1\ \mathit{Env}\ N\ E\ K, V), H \rightsquigarrow \text{Eval}(E, \mathit{Env}[N := V], K), H'}$$

$$\frac{\text{Lookup}(K, H) = \mathit{KCell} \qquad \text{Alloc}(\text{KProd}_1\ V\ \mathit{Env}\ K, H) = (P', H')}{\text{Apply}(\text{KProd0}\ \mathit{Env}\ R\ K, V), H \rightsquigarrow \text{Apply}(\mathit{KCell}, P'), H'}$$

$$\frac{\text{Lookup}(K, H) = \mathit{KCell} \qquad \text{Alloc}(\text{VProd}\ L\ V, H) = (P, H')}{\text{Apply}(\text{KProd}_1\ L\ K, V), H \rightsquigarrow \text{Apply}(\mathit{KCell}, P), H'}$$

$$\frac{\text{Lookup}(V, H) = (\text{VProd}\ X\ Y)}{\text{Apply}(\text{KZro}\ K, V), H \rightsquigarrow \text{Apply}(\mathit{KLookup}\ K, X), H'}$$

$$\frac{\text{Lookup}(V, H) = (\text{VProd}\ X\ Y)}{\text{Apply}(\text{KFst}\ K, V), H \rightsquigarrow \text{Apply}(\mathit{KLookup}\ K, Y), H'}$$

$$\frac{\text{Lookup}(K, H) = \mathit{KCell} \qquad \text{Alloc}(\text{VLeft}\ V, H) = (P', H')}{\text{Apply}(\text{KLeft}\ K, V), H \rightsquigarrow \text{Apply}(\mathit{KCell}, P'), H'}$$

$$\frac{\text{Lookup}(K, H) = \mathit{KCell} \qquad \text{Alloc}(\text{VRight}\ V, H) = (P', H')}{\text{Apply}(\text{KRight}\ K, V), H \rightsquigarrow \text{Apply}(\mathit{KCell}, P'), H'}$$

$$\frac{\text{Lookup}(V, H) = \text{VLeft}\ V'}{\text{Apply}(\text{KCase}\ \mathit{Env}\ \mathit{LN}\ L\ \mathit{RN}\ R\ K, V), H \rightsquigarrow \text{Eval}(L, \mathit{Env}[\mathit{LN} := V'], K), H}$$

$$\frac{\text{Lookup}(V, H) = \text{VRight}\ V'}{\text{Apply}(\text{KCase}\ \mathit{Env}\ \mathit{LN}\ L\ \mathit{RN}\ R\ K, V), H \rightsquigarrow \text{Eval}(R, \mathit{Env}[\mathit{RN} := V'], K), H}$$

Fig. 8. Abstract Machine Transition: Apply

tock tree will also store its machine state. To recompute an evicted value at some tock i, the tock tree will find the largest machine 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 machine 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 machine state *afterwards*, at step $t + 2$. A transition might not alloc any new cells, in such

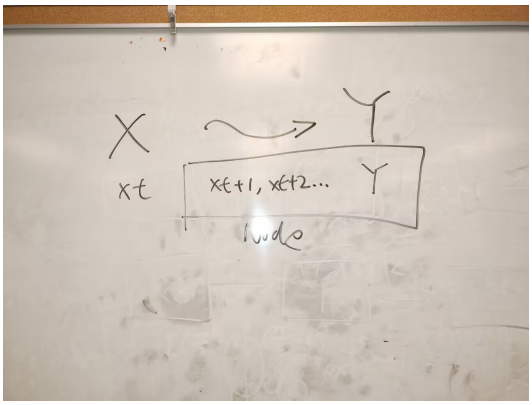

Fig. 9. a node in the tock tree

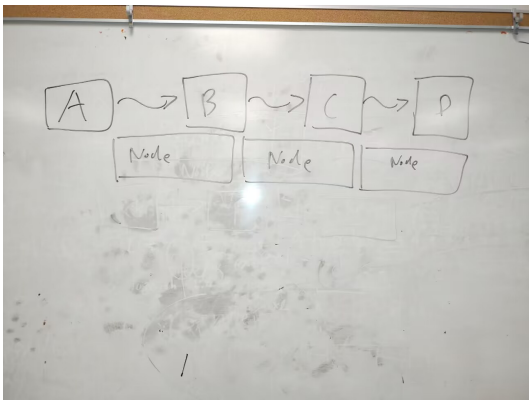

Fig. 10. the tock tree with multiple nodes

a case the Node only store a machine state any no memory cell. To be more specific, if the node represent the execution of the CEK machine at step t, it will only contain the CEK machine state at step $t + 1$.

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 Happy path

Every state transition and pointer lookup is then converted into node insertion and node lookup into this data structure. After we had completed a transition, we construct the node, packing the allocated cells during the transition and the transit-to state, inserting the node onto the tock tree.

Originally, transition might require pointers lookup. Such lookup is translated to a query to the tock tree with the given node. Ideally speaking, the returned node contain the cell that correspond to the given tock. We can then convert the tock into the corresponding cell and continue execution. Note that we still need the tock tree to be lenient in this case, as the key of the node denote the beginning of the transition, not the cell itself.

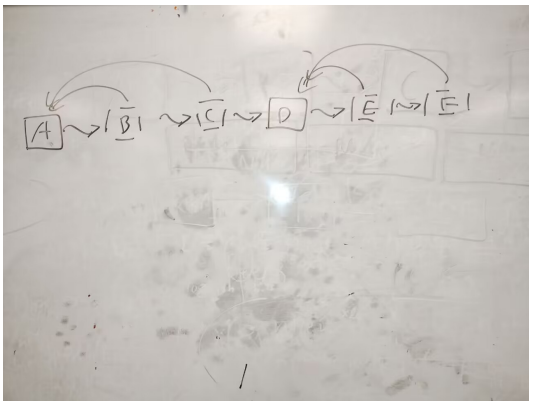

Fig. 11. lookup failure return the latest earlier node

4.4 Sad Path

Sadly, as we had removed nodes from the Tock Tree, we might not be able to retrieve a cell directly, and might need to recompute it - the point of the paper.

To recompute a value at tock t, the following steps are taken:

(1) Suspend the current execution into another kind of continuation, Replay Continuation (RK).
(2) Execute the transit-to state from the lookuped node in the tock tree, until tock reach t.
(3) Resume RK with the cell created at tock t.

Note that the Replay Continuation operate at a more fine-grain granularity then that of the normal continuation. This is because a single transit step might do multiple lookup, but RK need to correspond to a lookup failure in such a transition. Otherwise the requested value might be immediately uncomputed again, and the whole process enter an infinite loop.

After the above 3 steps, the execution shall continue as if no replay had happens at all, and lookup return a node with the cell we wanted. In other words, the happy path and the sad path should converge.

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

4.5 Reified Continuation

We treat continuations also as values, and label them with tock/put them in the tock tree, just like any other values. This allow us to also uncompute continuation as well.

4.6 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.

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

Fig. 12. Tock Tree API. Note that we deliberately avoid dictating what node get uncomputed, in order to decouple uncomputation/recomputation with selecting what to uncompute. Instead, node might be dropped during insertion into TockTree. One might e.g. set a limit onto the amount of nodes in the Tock Tree.

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

5 CEKR MACHINE

In this section we formalize the semantic of the language with uncomputation and recomputation. It is implemented by adding a Replay Continuation alongside the CEK Machine. Hence we call it the CEKR Machine.

5.1 Tock Tree

5.2 Replay Continuation

6 IMPLEMENTATION

6.1 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 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.

6.2 Picking Uncomputation Candidate

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

6.2.1 Union Find.

6.2.2 The Policy.

$$\frac{}{(\text{State}|\mathit{Return\ Cell}), rk, \mathit{TockTree} \rightsquigarrow (\text{State}|\mathit{Return\ Cell}), rk, \mathit{TockTree}}$$

$$\frac{}{\mathit{CheckReturn}(t, (\text{Nothing}, st), rk) = st} \qquad \frac{t + 1! = t'}{\mathit{CheckReturn}(t, (\_, st), \text{NoReplay}) = st}$$

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

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

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

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

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

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

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

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

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

Fig. 13. CEKR transition: Return

6.2.3 GDSF.

6.3 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.

6.4 Optimization

6.4.1 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,

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

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

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

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

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

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

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

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

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

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

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

Fig. 14. CEKR Transition: Eval

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.

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

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

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

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

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

$$\frac{Query(tt, V) = (V - 1, (\mathsf{Clos}Env'NE), \_) \qquad st = \mathsf{Eval}(X, Env, t + 1, t + 2) \qquad node = (\mathsf{Just}KApp1Env'NEK'}{\mathsf{Apply}(\mathsf{KApp0}\ Env\ X\ K', V, t), rk, tt \rightsquigarrow PostProcess(t, node, rk, tt)}$$

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

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

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

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

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

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

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

Fig. 15. CEKR Transition: Apply

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

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

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

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

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

Fig. 16. CEKR Transition: Apply

6.5 Bit counting

7 FORMAL GUARANTEE

7.1 Correctness

7.1.1 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.

With query_ghost, we can compare values between CEK and CEKR. More specifically, a value X under Heap H in CEK is equal to a value Y under Tock T in CEKR, if the cell pointed to by X (via lookup) and pointed to by Y (via query_ghost) is equal. This is a recursive definition as cells might contain more values.

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.

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.

7.1.2 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.

Lemma: 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.

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.

7.2 Performance

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

8 EVALUATION

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