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), 9 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRO

2 OVERVIEW

FIGURE draw attention to KEY IDEA

explain argument key steps dependency between steps focus on the dependency

Section: CEK Machine: pointers, lookup, allocate substeps (2.5-3pg long) Figure: draw on whiteboard, take screenshot.

Section: Tocks (5pg long) tock: exploit linear property of CEK machine talk about time instead of memory mapping stored in an ordered tree Context (not a section) map is sparse lookup fail, need to recompute also store CEK context

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)

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

Name	N	::=	A set of distinct names
Expr	E	::=	N | Let N E E | Lam N E | App E E | Prod E E | Zro E | Fst E | Left E | Right E | Case E N E N E

Fig. 1. The source language

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

Fig. 2. Heap API

3 CORE LANGUAGE

Zombie works on a untyped, purely functional, call by value language. Program in the language is then executed by the CEK abstract machine.

The CEK machine is an transition system between states. In other words, given a state, the CEK machine formalize what that state might transit into. Running a program under the CEK machine then correspond to transiting the machine until it reach a non-transitable state.

A state in the CEK machine is consisted of 3 parts:

(1) Control, the expression currently being evaluated.
(2) Environment, a name-key map of free variable of Control.
(3) Kontinuation, which will be invoked when Control is evaluated.

Our formalization of the CEK machine alternate between two mode.

It start with the step mode, which break down a complex expression, focusing on a part of it, storing the other parts onto the continuation, until it find an atomic expression, then convert the atomic expression into a value, and call the other mode, apply.

The apply mode transform the values according to the continuation, giving back control to step once there is more expression to evaluate.

The machine start with a special continuation, Done, and end when Value V is being applied to the Done continuation. This denote that the original expression evaluate to value V.

We had deliberately chosen to represent our semantic by the CEK machine, as it have three crucial properties:

(1) It is deterministic. Given any state, it can transit to at most 1 state.
(2) It is linear. An execution of a program can be characterized as a (possibly infinite) sequence the abstract machine transit through.
(3) Each step take a small, bounded amount of work, and especially for lookup/alloc.

Note how our implementation is different from a CEK Machine: In particular, we had made pointers and pointer lookup explicit, as we later have to abstract over and reason over them. Similarly, allocation is now explicit as well.

Below we sketch out the language and the cek machine. Note that this is the standard semantic - there is neither uncomputation nor replaying present. Uncomputation will be represented independently afterward.

Continuation	K	::=	$P\langle \text{KCell}\rangle$
KCell		::=	Done | KLet N Env E K | KApp_0 Env E K | KApp_1 V 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	::=	$P\langle \text{VCell}\rangle$
VCell		::=	Clos Env N E | VProd V V | VLeft V | VRight V
Environment	Env	::=	$(N, V)\ldots$
State		::=	Step E Env K | Apply K V

Fig. 3. Definitions for the CEK Machine

$$\frac{}{\text{State}, H \rightsquigarrow \text{State}, H} \qquad \frac{}{\text{Step}(N, Env, K), H \rightsquigarrow \text{Apply}(K, Env(N)), H}$$

$$\frac{\text{Alloc}(\text{KLeft } K, H) = (P, H')}{\text{Step}(\text{Left } X, Env, K), H \rightsquigarrow \text{Step}(X, Env, P), H'}$$

$$\frac{\text{Alloc}(\text{KRight } K, H) = (P, H')}{\text{Step}(\text{Right } X, Env, K), H \rightsquigarrow \text{Step}(X, Env, P), H'}$$

$$\frac{\text{Alloc}(\text{KProd}_0\ K\ R, H) = (P, H')}{\text{Step}(\text{Prod } L\ R, Env, K), H \rightsquigarrow \text{Step}(L, Env, P), H'}$$

$$\frac{\text{Alloc}(\text{KZro } K, H) = (P, H')}{\text{Step}(\text{Zro } X, Env, K), H \rightsquigarrow \text{Step}(X, Env, P), H'} \qquad \frac{\text{Alloc}(\text{KFst } K, H) = (P, H')}{\text{Step}(\text{Fst } X, Env, K), H \rightsquigarrow \text{Step}(X, Env, P), H'}$$

$$\frac{\text{Alloc}(\text{KCase } LN\ L\ RN\ R\ Env, H) = (P, H')}{\text{Step}(\text{Case } X\ LN\ L\ RN\ R, Env, K), H \rightsquigarrow \text{Step}(X, Env, P), H'}$$

$$\frac{\text{Alloc}(\text{KLet } A\ K\ C\ Env, H) = (P, H')}{\text{Step}(\text{Let } A\ B\ C, Env, K), H \rightsquigarrow \text{Step}(B, Env, P), H'}$$

$$\frac{\text{Alloc}(\text{KApp}_0\ K\ X, H) = (P, H')}{\text{Step}(\text{App } F\ X, Env, K), H \rightsquigarrow \text{Step}(F, P), H'} \qquad \frac{\text{Alloc}(\text{Clos } Env(\text{fv}) \cdots N\ E, H) = (P, H')}{\text{Step}(\text{Lam } N\ E, Env, K), H \rightsquigarrow \text{Apply}(K, P), H'}$$

Fig. 4. Abstract Machine Transition: Step

4 UNCOMPUTING AND RECOMPUTING

4.1 Tock

The key insight in zombie is to introduce an abstraction layer on pointers and on the heap, turning pointers, keys into memory space, into tocks, which is 64bit int, keys into logical time.

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

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

$$\frac{\text{Lookup}(P, H) = \text{KCase } Env\ LN\ L\ RN\ R\ K \qquad \text{Lookup}(V, H) = \text{VLeft } V}{\text{Apply}(P, V) \rightsquigarrow \text{Step}(L, Env(LN := V), K)}$$

$$\frac{\text{Lookup}(P, H) = \text{KCase } Env\ LN\ L\ RN\ R\ K \qquad \text{Lookup}(V, H) = \text{VRight } V}{\text{Apply}(P, V) \rightsquigarrow \text{Step}(R, Env(RN := V), K)}$$

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

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

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

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

$$\frac{\text{Lookup}(P, H) = \text{KLet } A\ Env\ C\ K}{\text{Apply}(P, V), H \rightsquigarrow \text{Step}(C, Env(A := V), K), H'}$$

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

$$\frac{\text{Lookup}(P, H) = \text{KApp}_1\ F\ K \qquad \text{Lookup}(F, H) = (\text{Clos } Env\ N\ E, H)}{\text{Apply}(P, V), H \rightsquigarrow \text{Step}(E, Env(N := V), K), H'}$$

Fig. 5. Abstract Machine Transition: Apply

Tock start at 0, and is increased by 1 on every transition step in the abstract machine, or on every allocation. This created a one-to-one mapping between each transition/lookup(event), with each tock less then the current tock. This had unveil the linear ordering between each events. In particular, a Value allocated during tock X can be recreated by re-executing a transition starting at tock Y < X, because the CEK machine is deterministic, and will replay faithfully what had happened.

Note that for any tock X value, there are, barring edge cases many tock Y < X transition that can replay said value. conversely, for any tock Y, there are multiple tock X > Y value it replay.

The one-to-many mapping between transition and values permit us to store asymptotically less metadata than the amount of allocated values.

4.2 Tock Tree

To exploit this linearity, the memory space is transformed into a global binary search tree, a Tock Tree, keyed by the tocks.

Each node in the Tock Tree correspond to a state transition. A node at tock t, representing the transition starting at tock t, contain an array of cells(value representations) it allocated, at tock t+1, t+2..., paired with the state it transit to. Note that it store the transit-to state, but not the transit-from state, for that state is useless: the transit-from state can recompute the allocated cells during the transition, but we had already stored it anyway!

Unlike binary search tree, which fail when a lookup does not find the precise match, the tock tree is lenient. On a lookup of key x, if said key is not in the tock tree, it will return the node with the largest key y < x instead. Intuitively, this represent returning the latest transition that can replay to the queried node.

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.

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. 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.

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.

Formally speaking,

$$before : Value = P < VCell > \qquad after : Value = Tock \qquad TockTree :???$$

$$query : (TockTree, Tock) - > (Tock, Node) \qquad insert : (TockTree, Node) - > TockTree$$

$$Node = ([KCell|VCell], State) \qquad before : State = StepExprEnvK|ApplyKV$$

$$after : State = StepExprEnvKTock|ApplyKVTock$$

5 CEKR MACHINE

During execution, the tock needed to be converted back to a Cell. It proceed as follow:

(1) to convert tock t to a Cell:
(2) query the tock tree on t to get a Node
(3) if the cell is in the array, return said Cell
(4) otherwise, issue a replay to t.

A replay t suspend the current machine state, replacing it with the State in the Node returned from tock tree, and executing until the tock reach t. The old state is then resumed with the Cell at t. Replay is recursive: a replay might issue lookup that require more replay.

$$ReplayContinuation = RK :=$$

$$NoReplay|RKApplyVTockRK|RKCaseTockEnvNENEKRK|RKZroTockKRK|RKFstTockKRK$$

$$Replay = R := (State, RK)$$

$$RApply : ((Tock, Node), RK)- > (State, RK)RApply((,(cells, st)), NoReplay) =$$

$$(st, NoReplay)RApply((t, (cells, st)), RKApplyvt'rk) = if t + 1 <= t' <$$

$$t + 1 + len(cells)then(Applyvcells[t' - t -$$

$$1]t', rk)else(st, RKApplyvt'rk)RApply((t, (cells, st)), RKCaset'EnvLNLRNRKRK) = if t + 1 <=$$

$$t' < t + 1 + len(cells)thenif cells[t' - t - 1] = VLeftXthen((StepLEnv[LN :=$$

$$X]K, t'), rk)elseif cells[t' - t - 1] = VRightYthen(StepREnv[RN :=$$

$$Y]Kt', rk)else(st, RKCaset'EnvLNLRNRK)RApply((t, (cells, st)), RKZrot'K) = if t + 1 <= t' <$$

$$t + 1 + len(cells)thenif cells[t' - t - 1] =$$

$$VProdXYthen(ApplyXKt', RK)else(st, RKZrot'K)RApply((t, (cells, st)), RKFstt'K) = if t + 1 <=$$

$$t' < t + 1 + len(cells)thenif cells[t' - t - 1] = VProdXYthen(ApplyYKt', RK)else(st, RKFstt'K)$$

$$FromTock : (TockTree, Tock)- >$$

$$VCell|KCell|StateFromThetocktree, trytogetthecorrespondingcell.Ifthecellisuncomputed, returntheclosests$$

$$FromTock(tt, t) = let(Nodecellsstate, nt) = query(tt, t)inifnt + 1 <= t <$$

$$nt + 1 + len(cells)thencells[t - nt - 1]elsestate$$

$$goto : (Replay, TockTree)- > (Replay, TockTree)State = StepExprEnvK|ApplyKV$$

$$Step(N, Env, K), Hgotoapply(K, Env(N)), H$$

$$Step(LeftX, Env, K), HgotoStep(X, Env, P), H'where(P, H') = Alloc(KLeftK, H)$$

$$Step(RightX, Env, K), HgotoStep(X, Env, P), H'where(P, H') = Alloc(KRightK, H)$$

$$Step(CaseXLNLRNR, Env, K), HgotoStep(X, Env, P), H'where(P, H') =$$

$$Alloc(KCaseLNLRNREnv, H)$$

$$Step(ProdLR, Env, K), HgotoStep(L, Env, P), H'where(P, H') = Alloc(KProd0KR, H)$$

$$Step(ZroX, Env, K), HgotoStep(X, Env, P), H'where(P, H') = Alloc(KZroK, H)$$

$$Step(FstX, Env, K), HgotoStep(X, Env, P), H'where(P, H') = Alloc(KFstK, H)$$

$$Step(LetA = BinC, Env, K), HgotoStep(B, Env, P), H'where(P, H') = Alloc(KLetAKCEnv, H)$$

$$Step(Appfx, Env, K), HgotoStep(f, P), H'where(P, H') = Alloc(KApp0Kx, H)$$

$$Step(LamNE, Env, K), HgotoApply(K, P), H'where(P, H') = Alloc(ClosEnv(fv)...NE, H)$$

$$Apply(P, V) = apply(Lookup(P), V)apply(KLeftK, V), HgotoApply(K, P), H'where(P, H') =$$

$$Alloc(VLeftV, H)$$

$$apply(KRightK, V), HgotoApply(K, P), H'where(P, H') = Alloc(VRightV, H)$$

$$apply(KCaseEnvLNLRNRK, V), HgotoifLookUp(V, H) = VLeftVthenStep(L, Env(LN :=$$

$$V), K) = VRightVthenStep(R, Env(RN := V), K)$$

$$apply(KProd0EnvRK, V), HgotoStep(R, Env, P), H'where(P, H') = Alloc(KProd1VEnvK, H)$$

$$apply(KProd1LK, V), HgotoApply(P, K), H'where(P, H') = Alloc(VProdLV, H)$$

$$apply(KZroK, V), HgotoApply(X, K), HwhereVProdXY = LookUp(V, H)$$

$$apply(KFstK, V), HgotoApply(Y, K), HwhereVProdXY = LookUp(V, H)$$

$$apply(KLetAEnvCK, B), HgotoStep(C, Env(A := B), K), H$$

$$apply(KApp0EnvXK, V)gotoStep(X, Env, P)where(P, H'), H' = Alloc(KApp1VK, H)$$

$$apply(KApp1FK, V), HgotoStep(E, Env(N := V), K), Hwhere(ClosEnvNE) = LookUp(V, H)$$

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.

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, 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.

6.5 Bit counting

7 FORMAL GUARANTEE

7.1 Safety

Evaluating under replay semantic give same result as under normal semantic

7.2 Liveness

Evaluating will eventually produce a value

Decreasing on lexicalgraphic ordering on the replay stack do work

7.3 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