Uncomputation

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

Program execution need memory. Program may run out of memory for multiple reasons: big dataset, exploding intermediate state, the machine have less memory than others, etc. When this happens, the program either get killed, or the operating system swaps, significantly degrading the performance. We propose a technique, uncomputation, that allow the program to continue running gracefully even after breaching the memory limit, without significant performance degradation. Uncomputation work by turning computed values back into thunk, and upon re-requesting the thunk, computing and storing them back. A naive implementation of uncomputation will face multiple problems. Among them, the most crucial and the most challenging one is that of breadcrumb. After a value is uncomputed, it's memory can be released but some memory, breadcrumb, is needed, so we can recompute the value back. 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, seemingly storing recompute information in 0-bits and violating information theory.

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

ACM Reference Format:
Maisa Kirisame and Pavel Panchekha. 2018. Uncomputation. *J. ACM* 37, 4, Article 111 (August 2018), 8 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: 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

Sad Path

Replay Stack

Double O(1)

Quality

Another kind of thing in tock tree.

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

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

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<X>	P<X>	::=	Key into heap with value type X
Alloc		:	(X, H) -> (Pointer<X>, H)
Lookup		:	(Pointer<X>, H) -> X

Fig. 2. Heap API

Continuation	K	::=	P<KCell>
KCell		::=	Stop | KLet N Env E K | KApp0 Env E K | KApp1 V K |
			KProd0 Env E K | KProd1 V K | KZro K | KFst K |
			KLeft K | KRight K | KCase Env N E N E K
Value	V	::=	P<VCell>
VCell		::=	Clos Env N E | VProd V V | VLeft V | VRight V
Environment	Env	::=	(N, V)…
State		::=	Step Expr Env K | Apply K V

Fig. 3. Definitions for the CEK Machine

Transition from X -> Y
Allocated during the transition
Map from tock to Cell | Context

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.

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

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

(1) Note how our implementation is different from a CEK Machine: In particular, we had made pointer and pointer lookup explicit, as we later have to abstract over and reason over them.

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

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.

$$\frac{}{\text{State}, \text{Heap} \rightsquigarrow \text{State}, \text{Heap}}$$

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

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

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

$$\frac{\text{Alloc}(\text{KProd0 K R}, \text{H}) = (\text{P}, \text{H'})}{\text{Step}(\text{Prod L R}, \text{Env}, \text{K}), \text{H} \rightsquigarrow \text{Step}(\text{L}, \text{Env}, \text{P}), \text{H'}}$$

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

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

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

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

$$\frac{\text{Alloc}(\text{KApp0 K X}, \text{H}) = (\text{P}, \text{H'})}{\text{Step}(\text{App F X}, \text{Env}, \text{K}), \text{H} \rightsquigarrow \text{Step}(\text{F}, \text{P}), \text{H'}}$$

$$\frac{\text{Alloc}(\text{Clos Env}(\text{fv}) \ldots \text{N E}, \text{H}) = (\text{P}, \text{H'})}{\text{Step}(\text{Lam N E}, \text{Env}, \text{K}), \text{H} \rightsquigarrow \text{Apply}(\text{K}, \text{P}), \text{H'}}$$

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

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

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

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

$$\frac{\text{Lookup}(\text{P}, \text{H}) = \text{KProd0 Env R K} \qquad \text{Alloc}(\text{KProd1 V Env K}, \text{H}) = (\text{P}, \text{H'})}{\text{Apply}(\text{P}, \text{V}), \text{H} \rightsquigarrow \text{Apply}(\text{K}, \text{P}), \text{H'}}$$

$$\frac{\text{Lookup}(\text{P}, \text{H}) = \text{KProd1 L K} \qquad \text{Alloc}(\text{VProd L V}, \text{H}) = (\text{P}, \text{H'})}{\text{Apply}(\text{P}, \text{V}), \text{H} \rightsquigarrow \text{Apply}(\text{K}, \text{P}), \text{H'}}$$

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

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

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

$$\frac{\text{Lookup}(P, H) = \text{KApp0 Env X K} \qquad \text{Alloc}(\text{KApp1 V K}, H) = (P, H')}{\text{Apply}(P, V), H \leadsto \text{Step}(X, \text{Env}, P), H'}$$

$$\frac{\text{Lookup}(P, H) = \text{KApp1 F K} \qquad \text{Lookup}(F, H) = (\text{Clos Env N E}, H)}{\text{Apply}(P, V), H \leadsto \text{Step}(E, \text{Env}(N := V), K), H'}$$

4 UNCOMPUTING AND RECOMPUTING

4.1 Tock

Number every step and introduce tock, sitting between memory and program execution instead of referring to point in space, refer to point in time.

Turn pointers of space to pointers of path

introduce an allocate step. allocation or normal transition advance the tock and get assigned a numbering

The critical insight of zombie is that multiple abstract machine state compute the same value. To be more precise, if a machine state x step to a machine state y, x must have computed all value that y might compute, and possibly more. This indicate that we do not have to store all previous machine state - some might be dropped in favor of older ones.

For this purpose, we introduced a global, logical time of 64 bit int, a tock. Tock start at 0, and increase by 1 on each transition(step/apply) in the abstract machine, and whenever a value is constructed. Conversely, all tock smaller then the current tock correspond to either a value, or a executed machine state, and vice versa.

More importantly, given a value that correspond to tock X, any machine state with tock Y < X will recompute it. The largest Y under the constraint will do the least amount of transition computing said value.

4.2 Tock Tree

To pair a value with it's tock concretely(I mean in the runtime, the word look bad), and to allow a value to be recomputed, we abstract over the memory space(need better words), replacing pointers to value, to tocks instead. The actual values are stored on a global data structure, the tock tree. Reading from a pointer is replaced from querying the tock tree with the tock. The tock tree additionally store machine state as they are executed, so a value might be uncomputed and recomputed in the future with any earlier machine state.

The tock tree is a binary search tree with the crucial property that lookup returns the largest node with key <= the input. This allow us to drop any node in the tock tree, with the exception of the leftmost node. Each node on the tock tree, at point t, correspond to an execution of a transition, that started at tock t, and contain:

(1) An array of cell(actual value), created during the transition, of corresponding tock t+1, t+2...
(2) The state it transit to.

Note that it store the transit-to state, but not the transit-from state, for that state is useless.

Uncomputing is then merely deleting a non-leftmost value from the tock tree.

Formally speaking,

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

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

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

$$after : State = StepExprEnvKTock|ApplyKVTock$$

4.3 Replay

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.

$$ReplayContinuation = RK :=$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$$Alloc(KCaseLNLRNREnv, H)$$

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

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

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

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

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

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

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

$$Alloc(VLeftV, H)$$

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

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

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

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

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

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

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

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

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

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

5 IMPLEMENTATION

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 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

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

5.2.1 Union Find.

5.2.2 The Policy.

5.2.3 GDSF.

5.3 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.

5.4 Optimization

5.4.1 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.

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

5.5 Bit counting

6 FORMAL GUARANTEE

6.1 Safety

Evaluating under replay semantic give same result as under normal semantic

6.2 Liveness

Evaluating will eventually produce a value

Decreasing on lexicalgraphic ordering on the replay stack do work

6.3 Performance

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

7 EVALUATION

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