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), 12 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 untyped, purely functional, call by value language give it a name so we can refer to it. 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:

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

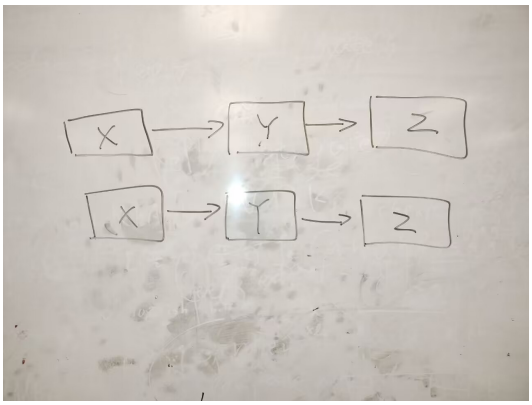

Fig. 1. the deterministic, linear nature of the CEK machine

(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.

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

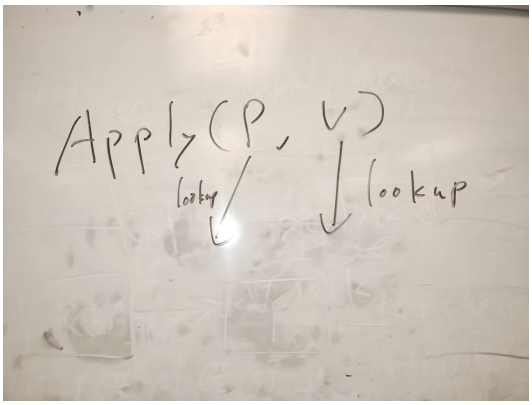

Fig. 2. the machine does a small constant amount of pointer lookup

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. 3. 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		:	$(X, H) \to (\text{Pointer}\langle X \rangle, H)$
Lookup		:	$(\text{Pointer}\langle X \rangle, H) \to X$

Fig. 4. Heap API

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		::=	Eval E Env K | Apply K V

Fig. 5. Definitions for the CEK Machine

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

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

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

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

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

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

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

$$\frac{\mathsf{Alloc}(\mathsf{KLet}\ A\ K\ C\ Env, H) = (P, H')}{\mathsf{Eval}(\mathsf{Let}\ A\ B\ C, Env, K), H \rightsquigarrow \mathsf{Eval}(B, Env, P), H'} \qquad \frac{\mathsf{Alloc}(\mathsf{KApp}_0\ K\ X, H) = (P, H')}{\mathsf{Eval}(\mathsf{App}\ F\ X, Env, K), H \rightsquigarrow \mathsf{Eval}(F, P), H'}$$

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

Fig. 6. Abstract Machine Transition: Step

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 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.

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

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

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

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

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

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

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

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

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

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

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

Fig. 7. Abstract Machine Transition: Apply

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 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.

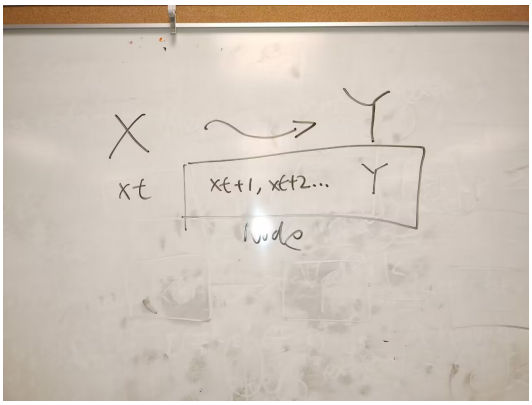

Fig. 8. a node in the tock tree

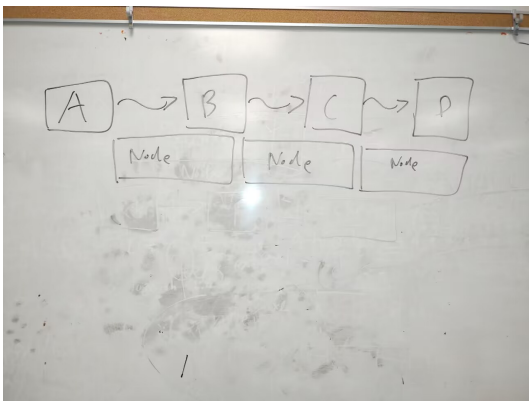

Fig. 9. the tock tree with multiple nodes

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 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.

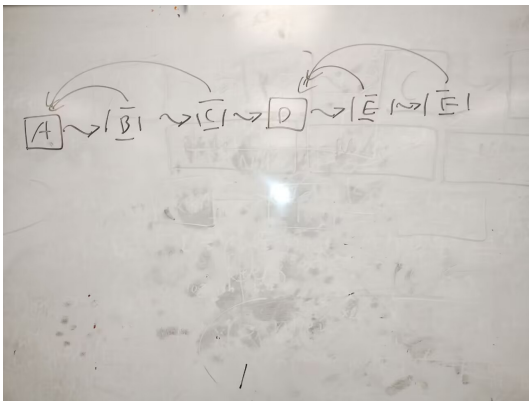

Fig. 10. lookup failure return the latest earlier node

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.

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

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			=	([KCell | VCell], State)
query	:	(TockTree, Tock)	->	(Tock, Node)
insert	:	(TockTree, Node)	->	TockTree

Fig. 11. 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 | RKApply V Tock RK | RKCase Tock Env N E N E K RK | RKZro Tock K RK | RKFst Tock K RK
Replay	R	::=	(State, RK)

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.

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.

$$\frac{}{RApply : (Tock, Node, RK) \rightarrow (State, RK)} \qquad \frac{}{RApply(\_, (\_, st), NoReplay) = (st, NoReplay)}$$

$$\frac{}{RApply(\_, (\text{Nothing}, st), rk) = (st, rk)} \qquad \frac{t + 1! = t'}{\text{RApply}((t, \text{Just } cell, st), rk) = (st, rk)}$$

$$\frac{t + 1 = t'}{\text{RApply}((t, \text{Just } cell, st), \text{RKApply } vt'rk) = (\text{Apply } vcellt', rk)}$$

$$\frac{t + 1 = t'}{\text{RApply}((t, \text{Just VLeft}X, st), RKCase\ t'\ Env\ LN\ L\ RN\ R\ K\ RK) = ((\text{Eval } L\ Env[LN := X]K, t'), rk)}$$

$$\frac{t + 1 = t'}{\text{RApply}((t, \text{Just VRight}X, st), RKCase\ t'\ Env\ LN\ L\ RN\ R\ K\ RK) = (\text{Eval } R\ Env[RN := Y]Kt', rk)}$$

$$\frac{t + 1 = t'}{\text{RApply}((t, \text{Just VProd } X\ Y, st), RKZro\ t'\ K\ RK) = (\text{Apply } X\ K\ t', RK)}$$

$$\frac{t + 1 = t'}{\text{RApply}((t, \text{Just VProd } X\ Y, st), RKFst\ t'\ K\ RK) = (\text{Apply } Y\ K\ t', RK)}$$

5.1 Tock Tree

5.2 Replay Continuation

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

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

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

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

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

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

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

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

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

$$Alloc(KCaseLNLRNREnv, H)$$

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

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

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

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

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

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

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

$$Alloc(VLeftV, H)$$

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

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

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

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

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

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

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

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

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

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

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