Uncomputation

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

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

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

ACM Reference Format:
Marisa Kirisame, Heng Zhong, Tiezhi Wang, Sihao Chen, and Pavel Panchekha. 2018. Uncomputation. *J. ACM* 37, 4, Article 111 (August 2018), 26 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRO

1.1 Motivation

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

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

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

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

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

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

At best, programmers today approach memory savings in an ad-hoc, case-by-case way: they examine their program and rewrite it to use less memory, possibly by using caching, lazy computation, or novel algorithms. However, these case-by-case solutions are not always available, since they depend on the details of the computation. Moreover, even when available, such solutions can require substantial programmer effort, including rewriting the program data flow to enable new algorithms or architectures that reduce memory usage.

We instead seek a generic and automatic solution to the problem of reducing program memory consumption. By generic, we mean a solution that is applicable to arbitrary programs in a given programming language. By automatic, we mean a solution that requires no programmer annotations or other modifications to the program itself. In other words, we seek a runtime system that would automatically reduce the memory consumption of a program. Moreover, we would like to reduce memory consumption below the level of live heap memory (unlike garbage collection) that adapts to fine-grained program behavior (unlike swapping.)

Zombie is a generic and automatic runtime system that reduces program memory consumption by evicting values from memory and recomputing them when necessary. That is, Zombie is a runtime for a simple purely-functional program language (which we call λ_Z) that pairs every value on the heap with the information necessary to recompute that value if it were evicted. Zombie can then evict any heap value at any point in time, thereby reducing program memory consumption, working below the level of live heap memory and adapting to fine-grained program behavior. In doing so, Zombie preserves the original program semantics, makes forward execution progress and is relatively efficient in its use of memory. Of course, the cost of replay is longer program runtime (since individual program steps may be replayed many times); in this way, Zombie introduces a space-time tradeoff, where programs run faster when more memory is available. That said, various optimizations, such as batching, a fast path for hot objects, and union find for fast eviction, allows Zombie to run at moderate overhead compared to a more traditional runtime.

Moreover, Zombie achieves its memory reduction without adding a commensurate memory overhead. That is, the amount of metadata stored by Zombie to enable recomputation is proportional to the total memory, and independent of how long the program has been running. To achieve this, Zombie stores replay points sparsely, so that replaying a given computation might involve replaying earlier computations as well, starting from whatever earlier replay point is available. The

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

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

sparse replay points allow Zombie to keep the total memory dedicated to replay limited while still enabling it to recompute any arbitrary value.

To evaluate Zombie, we test 10 λ_Z programs that implement common algorithmic tasks such as balanced binary trees, list and array manipulation, and recursive functions. Each program is run with different memory limits, ranging as low as one thousandth of a traditional run-time's memory consumption. Zombie allows the programs to run despite severe memory restrictions, with a 10× reduction in memory typically resulting in a 5× increase in running time.

In short, this paper contributions:

- Zombie, a runtime that saves memory for arbitrary programs without programmer effort;
- Proofs that Zombie preserves program semantics, makes progress, and is efficient;
- A collection of optimizations that enable Zombie to run with moderate runtime overhead.

2 OVERVIEW

The tock tree serves as a cache insert this sentence somewhere

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

Section: Implementation (3pg long) Heuristic Loop Unrolling

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

Summarize the meeting into a key step

Double O(1)

3 CEK MACHINES AND HEAPS

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

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

3.1 CEK Machine

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[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 $\mathsf{Apply}\ \mathsf{Done}\ V$.

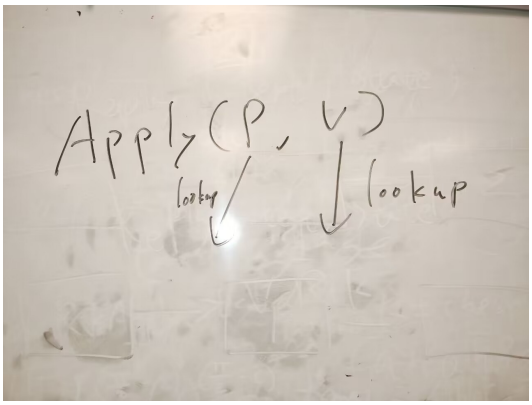

Fig. 3. Unlike a traditional small-step or big-step operational semantics, derivations (and thus computations) in the CEK machine form a flat list, with each step requiring a bounded amount of computation.

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

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

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

In such a small-step semantics, derivations form a tree, and the antecedent of the inference rule might require an arbitrarily-large derivation, and thus an arbitrary amount of computation. In the CEK machine, no such steps exist, and derivations in the CEK machine form a flat list. This property of the CEK machine is illustrated graphically in Figure 3.

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

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

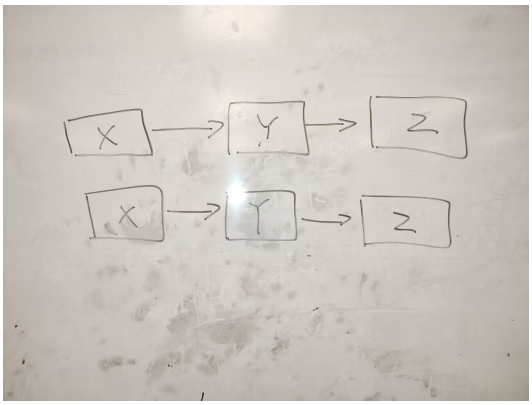

Fig. 4. Because the CEK machine is linear and deterministic, it can be "rewound" to a prior program point and be guaranteed to perform exactly the same sequence of states as during its prior execution.

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

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

3.2 Heap Memory

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

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

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

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

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

The full Eval and Apply transition relation is given in Appendix A.

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

4 UNCOMPUTING AND RECOMPUTING

4.1 Tock

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

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

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

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

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

4.2 Tock Tree

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

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

One thing worth noticing is that continuations are also reified. This means we treat them as like values, lifting pointers into tock, and storing continuations onto the tock tree. This allows us to also

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

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

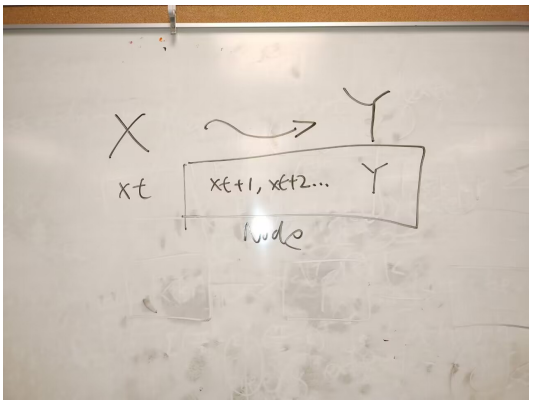

Fig. 7. a node in the tock tree

uncompute continuation as well, by evicting the node on the tock tree holding said continuation. This is crucial as traversing a deep data structure non-tail-recursively produces a continuation chain of equal depth. In contrast to a ordinary runtime, in which this is a slight but inevitable unfortunate, in our case, the continuation storage can become the bottleneck of the system.

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

4.3 Replay Continuation

When a query to the tock tree fail, meaning that the return node is not an exact match, and therefore does not include the requested tock, replay is needed. However, in order to replay, we need to store the current context, so we know which tock we are looking for, and how to resume after we found said value. We call said information a ReplayContinuation(RK). Since replay is recursive, ReplayContinuation form a stack.

The ReplayContinuation store three things:

(1) The tock we are looking for
(2) A ReplayHole, which is a state with a Hole. When the tock is found, it will be plugged into the Hole to continue execution.
(3) The rest of the ReplayContinuation.

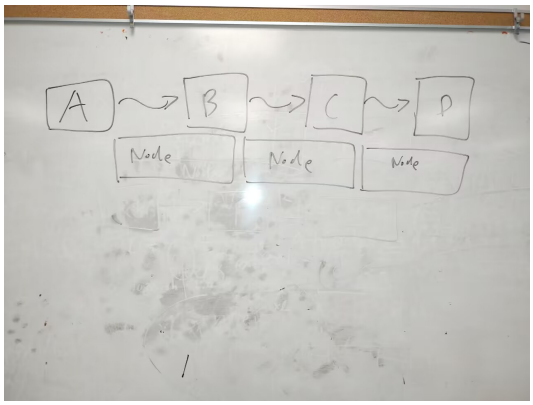

Fig. 8. the tock tree with multiple nodes

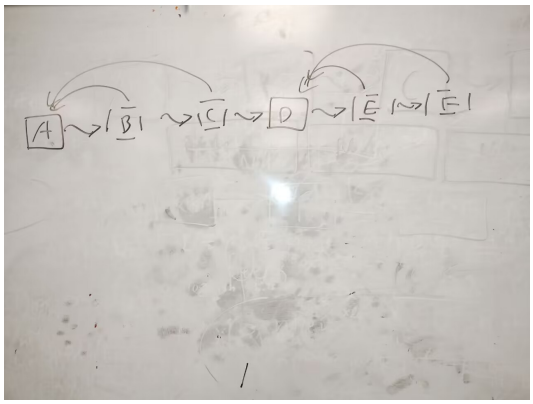

Fig. 9. lookup failure return the latest earlier node

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

4.4 CEKR Machine

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

The CEKR Machine consists of 6 parts:

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

(6) The tock tree

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

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

4.4.1 PostProcessing. The CEKR Machine executes by repeatedly computing the next node, and then applying a PostProcess step. the PostProcess step has two responsibilities: it inserts the node onto the tock tree, and checks if the node contains the cell the replay continuation is requesting. If so, it means the current transition has found the cell we are replaying for, and our ending state should be the Return state with said cell. Otherwise, it transits to the original state.

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

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

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

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

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

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

The transition is now split into 3 different cases:

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

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

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

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

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

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

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

Fig. 10. CEKR Transition: Returning

5 FORMAL GUARANTEE

In this section we prove that CEKR preserve(step to same value as CEK), progress(always step when CEK can step), and efficient.

5.1 Correctness

To prove that CEKR is correct (progress and preservation), we first define a transition on CEKR, GhostStepping. GhostStepping transit by assuming lookup always return the matching value, even if the value is evicted. Essentially, GhostStepping allow us to connect the CEKR machine with the CEK machine.

We then define another transition, OverStepping. A single step in OverStepping ignore all replay requested by lookup failure, continuing execution until the replays issued by the lookups had been resolved and returned. Our main theorem is a correspondence between GhostStepping and OverStepping, both of which happens on the CEKR Machine.

5.1.1 Tock Tree Setup. 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 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; Instead, each node is now paired with a boolean bit, indicating whether it has been evicted or not. We will then have a function QueryGhost : (TockTree, Tock) -> (Bool, Node) that returns the Node, along with a boolean indicating whether the node is present.

In some cases, we do not care about the eviction status of the tock tree, and we only care about the Node corresponding to each tock. For this purpose, we define two tock tree X, Y, to be 'equal modulo eviction', if they contains the same node, under different eviction status. Formally speaking,

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

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

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

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

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

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

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

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

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

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

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

Fig. 11. CEKR Transition: Eval

X eqmodev Y $\iff$ $\forall$ t, $\exists$ xev yev node LookupGhost(t, X) = (xev, node) and LookupGhost(t, Y) = (yev, node).

Using QueryGhost, we can define a function, LookupGhost, that return the Cell corresponding to a tock. We can then use LookupGhost to connect the CEKR machine to the CEK machine. With LookupGhost, we can connect the value from the CEKR machine to the CEK machine. More specifically, we can now define a Sem(standing for semantic) function, that converts a CEKR-value to a CEK-value under a given tock tree. The function works by recursively calling LookupGhost.

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 12. CEKR Transition: Apply

5.1.2 Ghost Stepping. Alongside QueryGhost, is the idea of GhostProduce and GhostStepping this name suck. GhostProduce is a function from (CEKR-State, TockTree) tuple to (Node, TockTree). GhostProduct represent executing a single transition via QueryGhost (so it need not involve any replay), returning the produced Node alongside the new TockTree, with the Node inserted.

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

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

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

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

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

Fig. 13. CEKR Transition: Apply

$$\frac{}{LookupGhost : (Tock, TockTree) \rightarrow CEK - Value | CEK - Continuation}$$

$$\frac{QueryGhost(t, tt) = (\_, Node(Justx, st))}{LookupGhost(t, tt) = x}$$

$$\frac{}{Sem : (CEKR - Value | CEKR - Continuation, TockTree) \rightarrow CEK - Value | CEK - Continuation}$$

$$\frac{LookupGhost(t, tt) = VProd(l, r)}{Sem(t, tt) = VProd(Sem(l, tt), Sem(r, tt))}$$

$$\frac{}{Convert : CEKR - State, TockTree \rightarrow CEK - State}$$

$$\frac{}{Convert(Eval(C, Env, K, t), tt) = Eval(C, Env[x = Sem(x, tt)...], Sem(K, tt))}$$

$$\frac{}{Convert(Apply(K, cell, t), tt) = Apply(Sem(K, tt), Sem(cell, tt))}$$

GhostProduct allow us to define GhostStepping, which is a transition from CEKR-Machine to CEKR-Machine, mimicing a CEKR-transition that cannot fail, as it use QueryGhost instead of Query. GhostStepping is unimplementable, but just like our formalization of Tock Tree and QueryGhost, it is purely for formalization purposes. As GhostStep does not need any replay, it does not have the replay continuation. GhostStep serves as a bridge between CEK and CEKR, connecting the two semantics. Once this is completed, we can only talk about correspondence between GhostStep and normal step, inside CEKR, without mentioning the CEK machine at all.

Lemma: CEK stepping is deterministic: ∀ CEK-State X Y Z, X $\rightsquigarrow$ Y and X $\rightsquigarrow$ Z $\implies$ X = Y. proof: trivial.

Lemma: Ghost stepping is deterministic: ∀ CEKR-Machine X Y Z, X $\rightsquigarrow$ Y and X $\rightsquigarrow$ Z $\implies$ X = Y. proof: trivial.

Lemma: Ghost stepping is congruent under eqmodev: ∀ (State | Return Cell) S S', ReplayContinuation RK RK', TockTree x y x', eqmodev x y and (S, RK), x $\rightsquigarrow$ (S', RK'), x' $\implies$ ∃ y', (S, RK), y $\rightsquigarrow$ (S', RK'), y' and eqmodev x' y'. proof: ghost stepping does not read from eviction status.

5.1.3 Well Foundness. A tock tree T is well-founded if:

(1) The leftmost node exists and is not evicted. (root-keeping)
(2) Forall node N inserted at tock T, N only refers to tock < T. (tock-ordering)
(3) Forall node pair L R (R come right after L in the tock tree), (L.state, T) GhostProduce R.
(4) Additionally, a (state, tock tree) pair is well-founded if the state is in the tock tree. (state-recorded)

Lemma: ghost stepping preserves 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 pairs except the last pair are irrelevant to the insertion because due to tock-ordering they cannot refer to it. For the last pair, the ghost-step must reach the same state due to determinism in ghost-stepping.
(4) state-recorded is maintained: we just inserted said state.

5.1.4 CEK and the CEKR. Theorem: under well-foundness, CEK-step and ghost-stepping preserve equivalence.

Formally: given CEK State X X', CEKR Machine ((s, NoReplay), tt), Convert(s, tt) = X and T is well-founded, X $\rightsquigarrow$ X', then: ((s, NoReplay), tt) $\rightsquigarrow$ ((s', NoReplay), tt'), such that Convert(s', tt') = X'.

is this theorem actually needed? or do we need a version with arbitrary replaycontinuation, which allow return as well? formalizing that is a bit hard

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

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

5.1.5 Over-Stepping. Our normal CEKR transition, however, does not one-to-one preserve with CEK transition, as it also include transition that failed and request Replay. To overcome this, we introduced Over-Stepping name suck, which take a step ignoring replay and what happens inside replay, in resemblance to Step-Over in debugger. Formally speaking, ((S, rk), tt) step to ((S', rk'), tt') if len(rk) >= len(rk'), ((S, rk), tt) step-star to ((S", rk"), tt"), with all len(rk") > len(rk), and ((S", rk"), tt") step to ((S', rk'), tt').

Theorem(Preservation): Under Well Founded Tock Tree, GhostStepping and OverStepping preserve.

Formally: if tt is WellFounded, ((S, rk), tt) GhostStep to ((S', rk'), tt'), and ((S, rk), tt) OverStep to ((S", rk"), tt"), then S' = S", rk' = rk", and tt' eqmodev tt". Proof: We induct on the number of transition OverStep actually make. It cannot be 0, so the base case start at 1. For the base case,

there can be no replay, which mean the query must succeed, and GhostStepping match this single CEKR-step. For the inductive case, where there are n+1 transition, the query must fail (otherwise the transition amount is 1, and it is the base case), so rk become rk_ = (Replaying t rh rk). ignoring the initial failing transition, the rest of the transition is a sequence of Over-Stepping that start with rk_, with the ending one decreasing rk_ back into rk, where the rest maintain rk_. With our inductive hypothesis, this sequence of Over-Stepping can be seen as a sequence of Ghost-Stepping. Then, by the well-foundness of the tock tree, this query-failure, followed by ghost-stepping preserving rk_, followed by a return ghost-stepping, is equivalent to a lookup that had not failed, so a single ghost stepping.

Theorem(Progress): Under Well Founded Tock Tree, OverStepping is not stuck if GhostStepping is not stuck.

Formally, if tt is WellFounded, ((S, rk), tt) OverStepping to M if ((S, rk), tt) GhostStep to M'.

Note that our Machine can get stuck as we are untyped. To overcome this, we piggyback the concept of progress on GhostStepping. Maybe our Preservation and Progress proof should be merged as one.

Proof:

If our query succeed, our theorem trivially hold, as OverStepping only consist of a single CEKR-step. Let's consider the case where it failed and request a replay to tock t. Let's called this ReplayContinuation we transit to rk. If overstepping is stuck, either there is infinitely many CEKR step with ending ReplayContinuation len >= len(rk), or CEKR is stuck during replay to tock t. But that CEKR-step had been played before, and our tock tree is well founded, so it is impossible. The only case that remain is the infinitely many CEKR step.

In such case, for each CEKR-Machine in the infinitely long transition, extract out all the request tock from the ReplayContinuation, forming a list of increasing tocks, and with the head of the list being the tock from the CEKR-State.

We can define a ordering on this list of tocks:

(1) $t : ts < ts$
(2) $a > b \rightarrow a : ts < b : ts$
(3) $x < y \rightarrow t : x < t : y$
(4) $a < b \wedge b < c \rightarrow a < c$

It is clear that this relation is transitive and antisymmetric. Additionally for any tock list ending with X, there are a finite amount of smaller elements ending with X, as the list must contain an increasing number of tocks.

However, each CEKR step in the infinite sequence must decrease on this list, and must also end with X (otherwise overstepping can produce and is not stuck), so we had reached a contradiction.

5.2 Performance

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

For each $v \in V$, we can

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

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

5.2.2 Converting CEKR Program into Pebble Game.

5.2.3 Theorem. Theorem: for a program that execute for N step, there exists a eviction strategy such that it consume at most O(n / log(n)) memory. Proof:

6 IMPLEMENTATION

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

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 grants frequently accessed data faster access time. Crucially, consecutive insertion takes amortized constant time.

The tock tree is then modified such that each node contains an additional parent and child pointer. The pointers form a list, which maintains a sorted representation of the tock tree. On a query, the tock tree does a binary search to find the innermost node, then follows the parent pointer if that node is greater than the key. This process is not recursive: the parent pointer is guaranteed to have a smaller node than the input key, as the binary search will yield the exact value, or the largest value less than the input, or the smallest value greater than the input.

6.2 IO

While our language is purely functional, we allow arbitrary IO effect directly. As IO is not referentially transparent, such effect cannot be replayed safely, and any node that executed a IO effect cannot be evicted, and therefore should not be considered for eviction.

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. The program is linked with mimalloc, with malloc and free being hooked to track the current memory consumption. Between every state transition, we check if the memory consumption is above a given memory limit. If so, we evict from the tock tree until it is no longer the case.

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 returned immediately. Otherwise, the default path is executed, and the weak reference is updated to point to the new Result.

6.4.2 Loop Unrolling. Each node in the tock tree is quite heavy. It contain multiple pointers (from the tock tree data structure), with the next state, which is a function pointers with a list of input tocks. As each node only store 0-1 objects, this pair each object with a high overhead. To combat this, and to avoid frequent insertion of node objects into tock tree, multiple state transition is packed into one. This mean that each node now store a vector of cells, with the next state unchanged. In our implementation, each node store 32 objects.

6.5 Picking Uncomputation Candidate

6.5.1 The Policy. In principle, we want to find the node that is the quickest to replay, take the most space, and is the stalest, to evict. More specifically, we want to find the node with the highest score = SpaceConsumption * TimeSinceLastAccess / ComputeTime.

To achieve this, for each node, we measure and store it's space consumption, time taken to execute, and last access time. Whenever a object is accessed, we update the corresponding node's last access time accordingly.

6.5.2 GD. As doing a linear search to find the node with the highest score is computationally infeasible, requiring a linear amount of work to free up a constant amount of memory, instead we adopt the greedy dual technique in caching.

Greedy dual is a technique in caching that approximate a eviction policy of X * staleness for arbitrary constant X. It's implementation consists of a priority heap alongside a value L. L represent roughly, the total amount of 'compute' that had been evicted. L start at 0, increasing by X whenever a value is evicted, and whenever an object is inserted into the cache, it have the priority of X + L.

6.5.3 Union Find. We want to avoid long, recursive replay, and instead prefer replay to complete as soon as possible. This suggest that our compute cost should be recursive: if replaying to node A require replaying node B, A's cost should also include B's cost. As tracking this require huge computational resource, we instead use a union find data structure to approximate recursive cost. On a high level, when node A and node B is evicted, and A depend on B, A and B should be in the same equivalent class.

7 EVALUATION

7.1 Machine

pavel - what is the spec of nightly?

7.2 Benchmark

Our benchmark consists of 10 program:

(1) taba, the 'there and back again' pattern discussed at overview.
(2) rbt, which implement a red black tree, and repeatedly insert into it.
(3) mergesum, which generate a list of number, then computing a new list of half the length by summing up two adjacent numbers. The process is repeated until a single number is computed.
(4) ski, a SKI Combinator calculus evaluator.
(5) list, create a list, then apply classical list manipulation transform (zip, map), on it.
(6) conv, create a list then apply convolution on it multiple time, and finally summing up all created list.
(7) conv2d, create a list of list then apply 2d convolution on it multiple time, summing up all created list like above.
(8) mergesort, create a list then sorting it.
(9) vector, using radix tree to create a vector, then updating the vector.
(10) fft, which use the vector implemented above, and implement the Fast Fourier Transform on it.

7.3 Evaluation Setup

Each program is tuned such that execution take 1-10 seconds (without any eviction).

Kontinuation	K	::=	$P\langle\text{KCell}\rangle$
KCell		::=	Done | $\mathit{KLookup}\ K$ | KLet $N\ E\ C\ K$ | $\text{KApp}_0\ E\ C\ K$ | $\text{KApp}_1\ E\ N\ C\ K$ | $\text{KProd}_0\ E\ C\ K$ | $\text{KProd}_1\ V\ K$ | KZro K | KFst K | KLeft K | KRight K | KCase $E\ N\ C\ N\ C\ K$
Value	V	::=	$P\langle\text{VCell}\rangle$
VCell		::=	Clos $E\ N\ E$ | VProd $V\ V$ | VLeft V | VRight V

Fig. 14. Value and Kontinuation definitions for the CEK-H Machine. Note that values and kontinuations are pointers, and the pointed-to value and kontinuation cells themselves contain pointers. As a result, all values on the heap are bounded-size.

We ran the programs with two different setup: A baseline, which does not include any tock tree data structure or eviction code, for comparison. And Zombie.

For Zombie, we execute the program once, without any eviction, and record down the highest memory consumption, alongside the runtime. We then execute the program with a memory limit being a decreasing ratio of said program, starting from 0.9 to 1e-4. When the program execution exceed 100x the zombie runtime without eviction, we consider it to be thrashing, and stopped executing said program, skipping all untried smaller ratio as well.

7.4 Result

7.5 Good Case Study: TABA

7.6 Bad Case Study: SKI

A CEK AND CEK-H MACHINE FORMALISM

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

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

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

$$\frac{}{\mathsf{Eval}(\backslash N.C, E, K) \rightsquigarrow \mathsf{Apply}(K, \mathsf{Clos}\ E(\mathrm{fv}) \cdots N\ C)} \qquad \frac{}{\mathsf{Eval}(F(X), E, K) \rightsquigarrow \mathsf{Eval}(F, \mathsf{KApp}_0\ K\ X)}$$

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

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

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

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

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

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

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

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

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

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

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

Fig. 15. Abstract Machine Transition: No Pointer

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

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

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

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

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

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

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

Fig. 16. Abstract Machine Transition: Eval

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 17. Abstract Machine Transition: Apply

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

Fig. 18. Definitions for the CEK Machine

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

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

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

$$\frac{}{\mathsf{Eval}(\backslash N.C, E, K) \rightsquigarrow \mathsf{Apply}(K, \mathsf{Clos}\ E(\mathrm{fv}) \cdots N\ C)} \qquad \frac{}{\mathsf{Eval}(F(X), E, K) \rightsquigarrow \mathsf{Eval}(F, \mathsf{KApp}_0\ K\ X)}$$

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

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

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

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

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

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

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

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

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

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

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

Fig. 19. Abstract Machine Transition: No Pointer

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

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

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

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

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

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

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

Fig. 20. Abstract Machine Transition: Eval

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 21. Abstract Machine Transition: Apply