

# 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 "tokens", 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 XXXX while incurring slowdowns of XXXX.

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), 24 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](mailto: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](mailto:hzhong21@m.fudan.edu.cn), Fudan University, 220 Handan Road, Yangpu District, Shanghai, P. R. China, 200437; Tiezhi Wang, [2152591@tongji.edu.cn](mailto:2152591@tongji.edu.cn), Tongji University, 4800 Caoan Road, Jiading District, Shanghai, P. R. China, 201804; Sihao Chen, [21201112@hdu.edu.cn](mailto: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.

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

© 2018 Copyright held by the owner/authors. Publication rights licensed to ACM.

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  $\lambda_Z$ ) 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	$C$	$::=$	$x \mid \text{Let } x = A \text{ in } B \mid \lambda x, A \mid F(A) \mid (A, B) \mid A.0 \mid$ $B.1 \mid \text{Left } A \mid \text{Right } B \mid \text{Case } A \text{ of } \{ \text{Left } x \Rightarrow$ $L \mid \text{Right } y \Rightarrow R \}$

Fig. 1. The syntax of  $\lambda_Z$ . 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  $\lambda_Z$  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 $\times$  reduction in memory typically resulting in a 5 $\times$  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 CEK 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  $\lambda_Z$ . For simplicity, we treat the language as untyped in this paper. The syntax of  $\lambda_Z$  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,  $\lambda_Z$  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) **C**ontrol, the expression currently being evaluated.
- (2) **E**nvironment, a map from the free variables of the Control to their values.

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

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) **K**ontinuation, 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:

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

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 ( $\text{Eval } C \ \{\} \ \text{Done}$ ), 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  $\text{Apply Done } V$ . In that state,  $V$  is the result of evaluating  $C$ .<sup>1</sup> 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:

$$\overline{\text{Eval}(F(X), E, K) \rightsquigarrow \text{Eval}(F, E, \text{KApp}_0 \ K \ X)}$$

Note that the argument  $X$  is stored in the  $\text{KApp}_1$  kontinuation, to be evaluated at a later point. The  $\text{KApp}_0$  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 ( $\text{Clos}$ ), at which point the CEK machine will transition to an Apply state, which will then transition to evaluating the argument:

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

Note that the components  $E'$ ,  $N$ , and  $C$  of the closure are now stored in the  $\text{KApp}_1$  kontinuation, which basically represents the one-hole context  $\lambda v, (\text{Clos } E' \ N \ C)(v)$ .

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

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

<sup>1</sup>Note that, because  $\lambda_Z$  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  $\text{Apply Done } V$ .

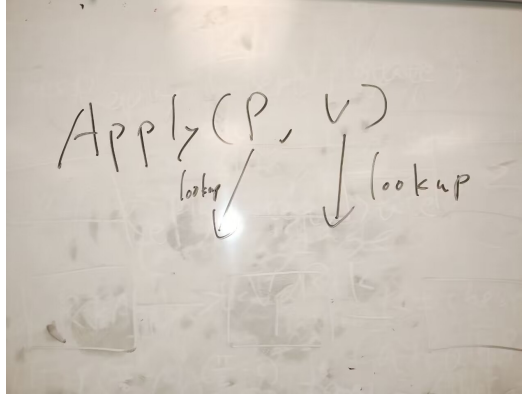


Fig. 3. Unlike a traditional small-step or big-step operational semantics, derivations (and thus computations) in the CEK machine form a flat list, with each step requiring a bounded amount of computation.

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:

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

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  $S_1 \rightarrow S_2$  with an increment  $t, S_1 \rightarrow t + 1, S_2$ . 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.

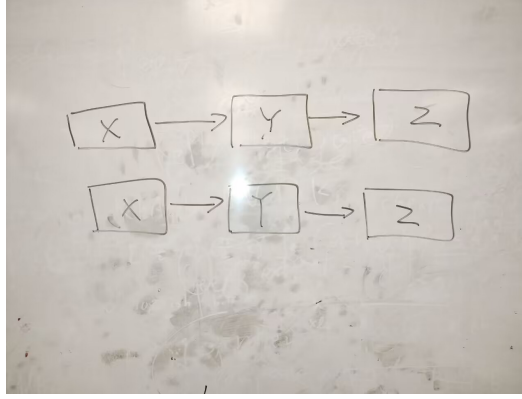


Fig. 4. Because the CEK machine is linear and deterministic, it can be “rewound” to a prior program point and be guaranteed to perform exactly the same sequence of states as during its prior execution.

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

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  $\lambda_Z$  program, we now extend the CEK machine with an explicit heap and explicit pointers. Values  $V$  are now implemented as pointers  $P\langle VCell \rangle$  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  $P\langle KCell \rangle$  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  $\text{KApp}_0$  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  $\text{KApp}_0$  kontinuation requires dereferencing the value pointer and then allocating the  $\text{KApp}_1$  kontinuation:

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

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	=	$P\langle VCell \rangle$	Tock
Continuation	=	$P\langle KCell \rangle$	Tock
State	::=	$Eval\ C\ E\ K \mid Apply\ K\ V$	$Eval\ C\ E\ K\ Tock \mid Apply\ K\ V\ Tock$
Node	=		$(Maybe(KCell \mid VCell), State)$
Query	:	$(TockTree, Tock)$	$\rightarrow (Tock, Node)$
Insert	:	$(TockTree, Tock, Node)$	$\rightarrow TockTree$

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.

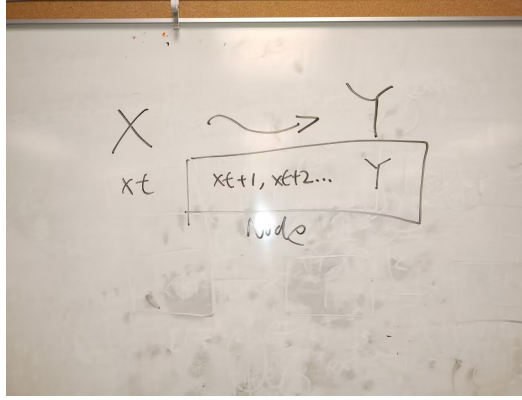


Fig. 7. a node in the tock tree

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 an 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

In order to replay for a cell at a tock  $t$ , we need to store (1) the tock  $t$  that we replay to, so we can stop once the value corresponding to  $t$  is found and (2) the original state before replaying, resume execution at that point, so we can stop once  $t$  is found, and the original state before replaying, so we can resume execution at that point. This is essentially a continuation, which we call the replay continuation (RK).

Note that it is different from the normal continuation in the following way:

- (1) continuation is an expr with a hole of type `expr`, while replay continuation is a state with a hole of type `cell`
- (2) there are fewer replay continuation types than continuation: this is because only the state that lookup a cell needs a replay continuation; a state that does not, does not need a replay,



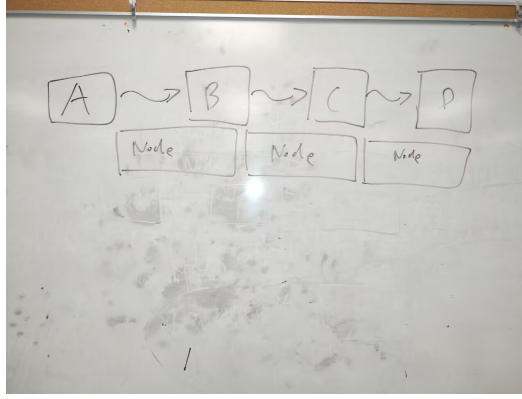


Fig. 8. the tock tree with multiple nodes

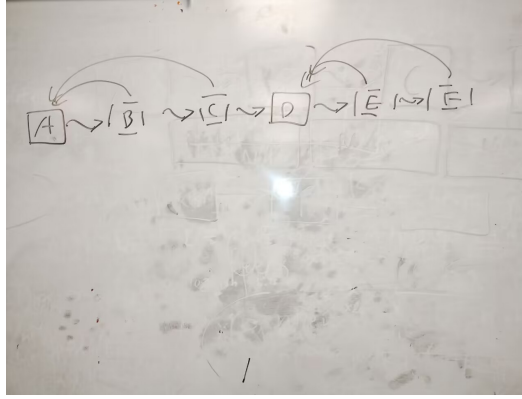


Fig. 9. lookup failure return the latest earlier node

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)

and so does not need replay continuation. An example is Eval of a function application, as it only focuses on the function, pushing the arguments onto the continuation stack.

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

#### 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) **C**ontrol, the expression currently being evaluated.
- (2) **E**nvironment, a map from the free variables of the Control to their values.
- (3) **K**ontinuation, which is to be invoked with the value the Control evaluates to.
- (4) **R**eplay, 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.

Machine ::= (State |Return Cell), rk, 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.

$$\begin{array}{c}
 \hline
 \text{CheckReturn} : (t, \text{node}, rk) \rightarrow \text{State}|\text{Returncell} \qquad \hline
 \text{CheckReturn}(t, (\text{Nothing}, st), rk) = st \\
 \\
 \hline
 t + 1! = t' \\
 \hline
 \text{CheckReturn}(t, (\_, st), \text{NoReplay}) = st \\
 \\
 \hline
 \text{CheckReturn}(t, (\text{Just cell}, st), \text{Replaying}(t + 1)rk) = \text{Return cell} \\
 \\
 \hline
 t + 1! = t' \\
 \hline
 \text{CheckReturn}(t, (\text{Just cell}, st), \text{Replaying}t'rk) = st \\
 \\
 \hline
 \text{PostProcess} : (t, cell, rk, \text{TockTree}) \rightarrow \text{Machine} \\
 \\
 \hline
 \text{PostProcess}(t, \text{node}, rk, tt) = \text{CheckReturn}(t, \text{node}, rk), rk, \text{insert}(tt, t, \text{node})
 \end{array}$$

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

$$\begin{array}{c}
\frac{st = \text{Apply}(\text{cell}, v, (t + 1)) \quad node = (\text{Nothing}, st)}{\text{Return } \text{cell}, \text{Replaying\_}(\text{RHLookup } v \ t)rk, tt \rightsquigarrow \text{PostProcess}(t, node, rk, tt)} \\
\\
\frac{st = \text{Eval}(L, E[LN := X], K, (t + 1)) \quad node = (\text{Nothing}, st)}{\text{Return } (\text{VLeft}X), \text{Replaying\_}(\text{RHCase } E \ LN \ L \ RN \ R \ K \ t), tt \rightsquigarrow \text{PostProcess}(t, node, rk, tt)} \\
\\
\frac{st = \text{Eval}(R, E[RN := Y], K, (t + 1)) \quad node = (\text{Nothing}, st)}{\text{Return } (\text{VRight}Y), \text{Replaying\_}(\text{RHCase } E \ LN \ L \ RN \ R \ K \ t), tt \rightsquigarrow \text{PostProcess}(t, node, rk, tt)} \\
\\
\frac{st = \text{Apply}(K, X, (t + 1)) \quad node = (\text{Nothing}, st)}{\text{Return } (\text{VProd } X \ Y), \text{Replaying\_}(\text{RHZro } K \ t), tt \rightsquigarrow \text{PostProcess}(t, node, rk, tt)} \\
\\
\frac{st = \text{Apply}(K, Y, (t + 1)) \quad node = (\text{Nothing}, st)}{\text{Return } (\text{VProd } X \ Y), \text{Replaying\_}(\text{RHFst } K \ t), tt \rightsquigarrow \text{PostProcess}(t, node, rk, tt)} \\
\\
\frac{st = \text{Eval}(X, E, (t + 1), (t + 2)) \quad node = (\text{Just}(\text{KApp1E}'NCK), st)}{\text{Return } (\text{ClosE}' \ N \ C), \text{Replaying\_}(\text{RHApp } E \ X \ K \ t), tt \rightsquigarrow \text{PostProcess}(t, node, rk, tt)}
\end{array}$$

Fig. 10. CEKR Transition: Returning

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.

## 5 IMPLEMENTATION

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

### 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 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.

### 5.2 Picking Uncomputation Candidate

### 5.3 Eviction

This allows us to remove any non-leftmost node from the tock tree. After the removal, the query that originally returned the removed node will return the node slightly earlier than that, which we can then replay to regenerate the removed node. In fact, this is the implementation of uncomputation

$$\begin{array}{c}
\frac{Query(tt, K) = (K - 1, Justkcell) \quad st = Apply(kcell, N, t + 1) \quad node = (Nothing, st)}{Eval(N, E, K, t), rk, tt \rightsquigarrow PostProcess(t, node, rk, tt)} \\
\\
\frac{Query(tt, K) = (X, st) \quad X! = K - 1}{Eval(N, E, K, t), rk, tt \rightsquigarrow st, Replaying K(RHLookup N t)rk, tt} \\
\\
\frac{st = Eval(X, E, t + 1, t + 2) \quad node = (Just KLet A K C E, st)}{(Eval(Let A B C, E, K, t), rk), tt \rightsquigarrow PostProcess(t, node, rk, tt)} \\
\\
\frac{st = Eval(X, E, t + 1, t + 2) \quad node = (Just Clos E(fv) \cdots N C, st)}{(Eval(Lam N C, E, K, t), rk), tt \rightsquigarrow PostProcess(t, node, rk, tt)} \\
\\
\frac{st = Eval(X, E, t + 1, t + 2) \quad node = (Just KApp_0 E X K, st)}{(Eval(App F X, E, K, t), rk), tt \rightsquigarrow PostProcess(t, node, rk, tt)} \\
\\
\frac{st = Eval(X, E, t + 1, t + 2) \quad node = (Just KProd_0 K R, st)}{(Eval(Prod L R, E, K, t), rk), tt \rightsquigarrow PostProcess(t, node, rk, tt)} \\
\\
\frac{st = Eval(X, E, t + 1, t + 2) \quad node = (Just KZro K, st)}{(Eval(Zro X, E, K, t), rk), tt \rightsquigarrow PostProcess(t, node, rk, tt)} \\
\\
\frac{st = Eval(X, E, t + 1, t + 2) \quad node = (Just KFst K, st)}{(Eval(Fst X, E, K, t), rk), tt \rightsquigarrow PostProcess(t, node, rk, tt)} \\
\\
\frac{st = Eval(X, E, t + 1, t + 2) \quad node = (Just KLeft K, st)}{(Eval(Left X, E, K, t), rk), tt \rightsquigarrow PostProcess(t, node, rk, tt)} \\
\\
\frac{st = Eval(X, E, t + 1, t + 2) \quad node = (Just KRight K, st)}{(Eval(Right X, E, K, t), rk), tt \rightsquigarrow PostProcess(t, node, rk, tt)} \\
\\
\frac{st = Eval(X, E, t + 1, t + 2) \quad node = (Just KCase LN L RN R E, st)}{(Eval(Case X LN L RN R, E, K, t), rk), tt \rightsquigarrow PostProcess(t, node, rk, tt)}
\end{array}$$

Fig. 11. CEKR Transition: Eval

in our system, and any non-leftmost node can be removed, to save memory at any given time. **move to implementation section**

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

### 5.3.1 Union Find.

### 5.3.2 The Policy.

### 5.3.3 GDSF.

$$\begin{array}{c}
\frac{Query(tt, K) = (X, (\_, st)) \quad X! = K - 1}{Apply(KLookupK, V, t, rk), tt \rightsquigarrow st, \text{Replaying } K \text{ (RHLookupVt)}rk, tt} \\
\\
\frac{Query(tt, K) = (K - 1, (Justcell, \_)) \quad st = Apply(cell, V, t + 1) \quad node = (Nothing, st)}{Apply(KLookupK, V, t), rk, tt \rightsquigarrow PostProcess(t, node, rk, tt)} \\
\\
\frac{st = Eval(C, Env[A := V], K', t + 1) \quad node = (Nothing, st)}{Apply(KLet A E C K', V, t), rk, tt \rightsquigarrow PostProcess(t, node, rk, tt)} \\
\\
\frac{Query(tt, V) = (X', (\_, st)) \quad X'! = V - 1}{(Apply(Just KApp0EnvXK, V, t), rk), tt \rightsquigarrow st, \text{Replaying } V \text{ (RHAppEnvXKt)}rk, tt} \\
\\
\frac{Query(tt, V) = (V - 1, (ClosEnv'NC), \_) \quad st = Eval(X, Env, t + 1, t + 2) \quad node = (JustKApp1E'NCK', s)}{Apply(KApp0 E X K', V, t), rk, tt \rightsquigarrow PostProcess(t, node, rk, tt)} \\
\\
\frac{st = Eval(C, Env[N := V], K', t + 1) \quad node = (Nothing, st)}{Apply(KApp1EnvNCK', V, t), rk, tt \rightsquigarrow PostProcess(t, node, rk, tt)} \\
\\
\frac{st = Eval(R, Env, t + 1, t + 2) \quad node = (JustKProd1VEnvK, st)}{Apply(KProd0EnvRK, V, t), rk, tt \rightsquigarrow PostProcess(t, node, rk, tt)} \\
\\
\frac{st = Apply(K, t + 1, t + 2) \quad node = (JustVProdLV, st)}{Apply(KProd1LK, V, t), rk, TT \rightsquigarrow PostProcess(t, node, rk, tt)} \\
\\
\frac{Query(tt, V) = (X, (\_, st)) \quad X! = V - 1}{Apply(KZroK, V, t), rk, tt \rightsquigarrow st, \text{Replaying } V \text{ (RHZroKt)}rk, tt} \\
\\
\frac{Query(tt, V) = (V - 1, (VProdXY, \_)) \quad st = Apply(K, X, t + 1) \quad node = (Nothing, st)}{Apply(KZroK, V, t), rk, tt \rightsquigarrow PostProcess(t, node, rk, tt)} \\
\\
\frac{Query(tt, V) = (X, (\_, st)) \quad X! = V - 1}{Apply(KFstK, V, t), rk, tt \rightsquigarrow st, \text{Replaying } V \text{ (RHFstKt)}rk, tt} \\
\\
\frac{Query(tt, V) = (V - 1, (VProdXY, \_)) \quad st = Apply(K, Y, t + 1) \quad node = (Nothing, st)}{Apply(KFstK, V, t), rk, tt \rightsquigarrow PostProcess(t, node, rk, tt)}
\end{array}$$

Fig. 12. CEKR Transition: Apply

#### 5.4 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.

$$\begin{array}{c}
\frac{st = \text{Apply}(K, t + 1, t + 2) \quad node = (\text{Just } VLeftV, st)}{\text{Apply}(KLeftK, V, t), rk, tt \rightsquigarrow \text{PostProcess}(t, node, rk, tt)} \\
\\
\frac{st = \text{Apply}(K, t + 1, t + 2) \quad node = (\text{Just } VRightV, st)}{\text{Apply}(KRightK, V, t), rk, tt \rightsquigarrow \text{PostProcess}(t, node, rk, tt)} \\
\\
\frac{Query(tt, V) = (X, (\_, st)) \quad X! = V - 1}{\text{Apply}(KCaseEnvLNLNRK, V, t), rk, tt \rightsquigarrow st, \text{Replaying}V(RHCaseEnvLNLNRKt)rk), tt} \\
\\
\frac{Query(tt, V) = (V - 1, (\text{Just } VLeftX, \_)) \quad st = \text{Eval}(L, E[LN := X], K', t + 1) \quad node = (\text{Nothing}, st)}{\text{Apply}(KCaseEnvLNLNRK, V, t), rk, tt \rightsquigarrow \text{PostProcess}(t, node, rk, tt)} \\
\\
\frac{Query(tt, V) = (V - 1, (\text{Just } VRightY, \_)) \quad st = \text{Eval}(R, E[RN := Y], K', t + 1) \quad node = (\text{Nothing}, st)}{\text{Apply}(KCaseEnvLNLNRK, V, t), rk, tt \rightsquigarrow \text{PostProcess}(t, node, rk, tt)}
\end{array}$$

Fig. 13. CEKR Transition: Apply

## 5.5 Optimization

**5.5.1 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.

**5.5.2 Loop Unrolling.** To avoid the frequent creation of node objects, and their insertion into the tock tree, multiple state transition is packed into one.

## 5.6 Bit counting

## 6 FORMAL GUARANTEE

### 6.1 Correctness

the CEK in this section does not use heap at all! In fact, I do not think we really need the heap version. I think we should rename the CEK with a pointer into CEKM, and explain it only for introductory purposes (or not at all).

**6.1.1 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  $\text{QueryGhost} : (\text{TockTree}, \text{Tock}) \rightarrow (\text{Bool}, \text{Node})$  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

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

$$\text{QueryGhost}(t, tt) = (\_, \text{Node}(\text{Just}x, st))$$

$$\text{LookupGhost}(t, tt) = x$$

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

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

$$\text{Sem}(t, tt) = \text{VProd}(\text{Sem}(l, tt), \text{Sem}(r, tt)) \quad \text{Convert} : \text{CEKR} - \text{Machine} \rightarrow \text{CEK} - \text{State}$$

modulo eviction', if they contains the same node, under different eviction status. Formally speaking,  $X \text{ eqmodev } Y \iff \forall t, \exists xev \text{ yev node } \text{LookupGhost}(t, X) = (xev, \text{node}) \text{ and } \text{LookupGhost}(t, Y) = (\text{yev}, \text{node})$ .

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.

**6.1.2 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. 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:  $\forall \text{CEK-State } X \ Y \ Z, X \rightsquigarrow Y \text{ and } X \rightsquigarrow Z \implies X = Y$ . proof: trivial.

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

Lemma: Ghost stepping is congruent under eqmodev:  $\forall (\text{State} \mid \text{Return Cell}) \ S \ S', \text{ReplayContinuation } RK \ RK', \text{TockTree } x \ y \ x', \text{eqmodev } x \ y \text{ and } (S, RK), x \rightsquigarrow (S', RK'), x' \implies \exists y', (S, RK), y \rightsquigarrow (S', RK'), y' \text{ and eqmodev } x' \ y'$ . proof: ghost stepping does not read from eviction status.

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

- (1) The leftmost node exists and is not evicted. (root-keeping)
- (2) For all node N inserted at tock T, N only refers to tock  $< T$ . (tock-ordering)
- (3) For all 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.

**6.1.4 Main Theorem.** Theorem: under well-foundness, CEK-step and ghost-stepping preserve equivalence.

Formally: given CEK State  $X$   $X'$ , CEKR Machine  $M$ , if  $\text{NoReplay}(M)$  and  $\text{Convert}(M) = X$  and  $T$  is well-founded,  $X \rightsquigarrow X'$ , and  $(M) \rightsquigarrow (M')$ ,  $\text{NoReplay}(M')$  and  $\text{Convert}(M') = X'$ .

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.

Lemma: stepping without changing  $\text{rk}$  is the same as ghost-stepping. Formally: if  $((S, \text{rk}), \text{tt}) \rightsquigarrow (S', \text{rk}, \text{tt}')$ ,  $((S, \text{rk}), \text{tt}) \text{GhostStepTo } ((S', \text{rk}), \text{tt}')$ . proof: trivial

Lemma: if stepping add to  $\text{rk}$ , the moment it changes back, is the same as single ghost-stepping

proof: due to well-foundness the cell  $\text{RApply}$  receives must be the same as the same cell on the tock tree but inaccessible. Note that the stepping made here cannot add a new node as it is bounded by the tock on the replay continuation, which is on the tock tree.

Lemma: it must change back

proof: by lexicographical ordering on  $\text{rk}$  and on state

Theorem:  $X, \text{rk}, \text{tt} \text{ step-star } Y, \text{rk}, \text{tt}' \rightarrow X \text{ tt ghost-step-star } Y, \text{tt}'$ , with  $\text{tt}' \text{ eqmodev } \text{tt}'$

proof: the step-star can be decomposed into a sequence of no-rk-change steps, or a push, with multiple steps, and finally a pop. either case is covered by a lemma above.

## 6.2 Performance

**6.2.1 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.

**6.2.2 Translation to pebble game.**

**6.2.3 Theorem.** memory consumption is linear to the number of object with  $O(1)$  access cost

## 7 EVALUATION

### A CEK AND CEK-H MACHINE FORMALISM



Kontinuation	$K$	$::= P\langle KCell \rangle$
KCell		$::= \text{Done} \mid KLookup\ K \mid KLet\ N\ E\ C\ K \mid$ $KApp_0\ E\ C\ K \mid KApp_1\ E\ N\ C\ K \mid KProd_0\ E\ C\ K \mid$ $KProd_1\ V\ K \mid KZro\ K \mid KFst\ K \mid KLeft\ K \mid$ $KRight\ K \mid KCase\ E\ N\ C\ N\ C\ K$
Value	$V$	$::= P\langle VCell \rangle$
VCell		$::= Clos\ E\ N\ E \mid VProd\ V\ V \mid VLeft\ V \mid VRight\ V$

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.

$$\begin{array}{c}
\frac{}{\text{State} \rightsquigarrow \text{State}} \qquad \frac{}{\text{Eval}(N, E, K) \rightsquigarrow \text{Apply}(K, E(N))} \\
\\
\frac{}{\text{Eval}(\text{Let } A = B \text{ in } C, E, K) \rightsquigarrow \text{Eval}(B, E, \text{KLet } A \text{ K } C \text{ } E)} \\
\\
\frac{}{\text{Apply}(\text{KLet } A \text{ E } C \text{ } K, V) \rightsquigarrow \text{Eval}(C, E[A := V], K)} \\
\\
\frac{}{\text{Eval}(\backslash N.C, E, K) \rightsquigarrow \text{Apply}(K, \text{Clos } E(\text{fv}) \cdots N \text{ } C)} \qquad \frac{}{\text{Eval}(F(X), E, K) \rightsquigarrow \text{Eval}(F, \text{KApp}_0 \text{ } K \text{ } X)} \\
\\
\frac{}{\text{Apply}(\text{KApp}_0 \text{ } E \text{ } X \text{ } K, \text{Clos } E' \text{ } N \text{ } C) \rightsquigarrow \text{Eval}(X, E, \text{KApp}_1 \text{ } E' \text{ } N \text{ } C \text{ } K)} \\
\\
\frac{}{\text{Apply}(\text{KApp}_1 \text{ } E \text{ } N \text{ } C \text{ } K, V) \rightsquigarrow \text{Eval}(C, E[N := V], K)} \quad \frac{}{\text{Eval}((L, R), E, K) \rightsquigarrow \text{Eval}(L, E, \text{KProd}_0 \text{ } K \text{ } R)} \\
\\
\frac{}{\text{Apply}(\text{KProd}_0 \text{ } E \text{ } R \text{ } K, V) \rightsquigarrow \text{Eval}(R, \text{Env}, \text{KProd}_1 \text{ } V \text{ } K)} \\
\\
\frac{}{\text{Apply}(\text{KProd}_1 \text{ } L \text{ } K, V) \rightsquigarrow \text{Apply}(K, \text{VProd } L \text{ } V)} \qquad \frac{}{\text{Eval}(X.0, E, K) \rightsquigarrow \text{Eval}(X, E, \text{KZro } K)} \\
\\
\frac{}{\text{Apply}(\text{KZro } K, \text{VProd } X \text{ } Y) \rightsquigarrow \text{Apply}(K, X)} \qquad \frac{}{\text{Eval}(X.1, E, K) \rightsquigarrow \text{Eval}(X, E, \text{KFst } K)} \\
\\
\frac{}{\text{Apply}(\text{KFst } K, \text{VProd } X \text{ } Y) \rightsquigarrow \text{Apply}(K, Y)} \qquad \frac{}{\text{Eval}(\text{Left } X, E, K) \rightsquigarrow \text{Eval}(X, E, \text{KLeft } K)} \\
\\
\frac{}{\text{Apply}(\text{KLeft } K, V) \rightsquigarrow \text{Apply}(K, \text{VLeft } V)} \qquad \frac{}{\text{Eval}(\text{Right } X, E, K) \rightsquigarrow \text{Eval}(X, E, \text{KRight } K)} \\
\\
\frac{}{\text{Apply}(\text{KRight } K, V) \rightsquigarrow \text{Apply}(K, \text{VRight } V)} \\
\\
\frac{}{\text{Eval}(\text{Case } X \text{ of Left } LN \Rightarrow L \parallel \text{Right } RN \Rightarrow R, E, K) \rightsquigarrow \text{Eval}(X, E, \text{KCase } LN \text{ } L \text{ } RN \text{ } R \text{ } E)} \\
\\
\frac{}{\text{Apply}(\text{KCase } E \text{ } LN \text{ } L \text{ } RN \text{ } R \text{ } K, \text{VLeft } V) \rightsquigarrow \text{Eval}(L, E[LN := V], K)} \\
\\
\frac{}{\text{Apply}(\text{KCase } E \text{ } LN \text{ } L \text{ } RN \text{ } R \text{ } K, \text{VRight } V) \rightsquigarrow \text{Eval}(R, E[RN := V], K)}
\end{array}$$

Fig. 15. Abstract Machine Transition: No Pointer

$$\begin{array}{c}
\frac{}{\text{State}, H \rightsquigarrow \text{State}, H} \qquad \frac{\text{Lookup}(K, H) = KCell}{\text{Eval}(N, E, K), H \rightsquigarrow \text{Apply}(KCell, E(N)), H} \\
\\
\frac{\text{Alloc}(KLet A K C E, H) = (P, H')}{\text{Eval}(\text{Let } A = B \text{ in } C, E, K), H \rightsquigarrow \text{Eval}(B, E, P), H'} \\
\\
\frac{\text{Lookup}(K, H) = KCell \quad \text{Alloc}(\text{Clos } E(\text{fv}) \cdots N C, H) = (P, H')}{\text{Eval}(\backslash N.C, E, K), H \rightsquigarrow \text{Apply}(KCell, P), H'} \\
\\
\frac{\text{Alloc}(KApp_0 K X, H) = (P, H')}{\text{Eval}(F(X), E, K), H \rightsquigarrow \text{Eval}(F, P), H'} \qquad \frac{\text{Alloc}(KProd_0 K R, H) = (P, H')}{\text{Eval}((L, R), E, K), H \rightsquigarrow \text{Eval}(L, E, P), H'} \\
\\
\frac{\text{Alloc}(KZro K, H) = (P, H')}{\text{Eval}(X.0, E, K), H \rightsquigarrow \text{Eval}(X, E, P), H'} \qquad \frac{\text{Alloc}(KFst K, H) = (P, H')}{\text{Eval}(X.1, E, K), H \rightsquigarrow \text{Eval}(X, E, P), H'} \\
\\
\frac{\text{Alloc}(KLeft K, H) = (P, H')}{\text{Eval}(\text{Left } X, E, K), H \rightsquigarrow \text{Eval}(X, E, P), H'} \qquad \frac{\text{Alloc}(KRight K, H) = (P, H')}{\text{Eval}(\text{Right } X, E, K), H \rightsquigarrow \text{Eval}(X, E, P), H'} \\
\\
\frac{\text{Alloc}(KCase LN L RN R E, H) = (P, H')}{\text{Eval}(\text{Case } X \text{ of Left } LN \Rightarrow L \parallel \text{Right } RN \Rightarrow R, E, K), H \rightsquigarrow \text{Eval}(X, E, P), H'}
\end{array}$$

Fig. 16. Abstract Machine Transition: Eval

$$\begin{array}{c}
\frac{\text{Lookup}(K, H) = KCell}{\text{Apply}(KLookup\ K, V), H \rightsquigarrow \text{Apply}(KCell, V), H'} \\
\\
\frac{}{\text{Apply}(KLet\ A\ E\ C\ K, V), H \rightsquigarrow \text{Eval}(C, E[A := V], K), H'} \\
\\
\frac{\text{Lookup}(V, H) = Clos\ E'\ N\ C \quad \text{Alloc}(KApp_1\ E'\ N\ C\ K, H) = (P', H')}{\text{Apply}(KApp_0\ E\ X\ K, V), H \rightsquigarrow \text{Eval}(X, E, P'), H'} \\
\\
\frac{}{\text{Apply}(KApp_1\ E\ N\ C\ K, V), H \rightsquigarrow \text{Eval}(C, E[N := V], K), H'} \\
\\
\frac{\text{Lookup}(K, H) = KCell \quad \text{Alloc}(KProd_1\ V\ E\ K, H) = (P', H')}{\text{Apply}(KProd_0\ E\ R\ K, V), H \rightsquigarrow \text{Apply}(KCell, P'), H'} \\
\\
\frac{\text{Lookup}(K, H) = KCell \quad \text{Alloc}(VProd\ L\ V, H) = (P, H')}{\text{Apply}(KProd_1\ L\ K, V), H \rightsquigarrow \text{Apply}(KCell, P), H'} \\
\\
\frac{\text{Lookup}(V, H) = (VProd\ X\ Y)}{\text{Apply}(KZro\ K, V), H \rightsquigarrow \text{Apply}(KLookup\ K, X), H'} \\
\\
\frac{\text{Lookup}(V, H) = (VProd\ X\ Y)}{\text{Apply}(KFst\ K, V), H \rightsquigarrow \text{Apply}(KLookup\ K, Y), H'} \\
\\
\frac{\text{Lookup}(K, H) = KCell \quad \text{Alloc}(VLeft\ V, H) = (P', H')}{\text{Apply}(KLeft\ K, V), H \rightsquigarrow \text{Apply}(KCell, P'), H'} \\
\\
\frac{\text{Lookup}(K, H) = KCell \quad \text{Alloc}(VRight\ V, H) = (P', H')}{\text{Apply}(KRight\ K, V), H \rightsquigarrow \text{Apply}(KCell, P'), H'} \\
\\
\frac{\text{Lookup}(V, H) = VLeft\ V'}{\text{Apply}(KCase\ E\ LN\ L\ RN\ R\ K, V), H \rightsquigarrow \text{Eval}(L, E[LN := V'], K), H} \\
\\
\frac{\text{Lookup}(V, H) = VRight\ V'}{\text{Apply}(KCase\ E\ LN\ L\ RN\ R\ K, V), H \rightsquigarrow \text{Eval}(R, E[RN := V'], K), H}
\end{array}$$

Fig. 17. Abstract Machine Transition: Apply

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

Fig. 18. Definitions for the CEK Machine

$$\begin{array}{c}
\frac{}{\text{State} \rightsquigarrow \text{State}} \qquad \frac{}{\text{Eval}(N, E, K) \rightsquigarrow \text{Apply}(K, E(N))} \\
\\
\frac{}{\text{Eval}(\text{Let } A = B \text{ in } C, E, K) \rightsquigarrow \text{Eval}(B, E, \text{KLet } A \text{ K } C \text{ } E)} \\
\\
\frac{}{\text{Apply}(\text{KLet } A \text{ E } C \text{ } K, V) \rightsquigarrow \text{Eval}(C, E[A := V], K)} \\
\\
\frac{}{\text{Eval}(\backslash N.C, E, K) \rightsquigarrow \text{Apply}(K, \text{Clos } E(\text{fv}) \cdots N \text{ } C)} \qquad \frac{}{\text{Eval}(F(X), E, K) \rightsquigarrow \text{Eval}(F, \text{KApp}_0 \text{ } K \text{ } X)} \\
\\
\frac{}{\text{Apply}(\text{KApp}_0 \text{ } E \text{ } X \text{ } K, \text{Clos } E' \text{ } N \text{ } C) \rightsquigarrow \text{Eval}(X, E, \text{KApp}_1 \text{ } E' \text{ } N \text{ } C \text{ } K)} \\
\\
\frac{}{\text{Apply}(\text{KApp}_1 \text{ } E \text{ } N \text{ } C \text{ } K, V) \rightsquigarrow \text{Eval}(C, E[N := V], K)} \quad \frac{}{\text{Eval}((L, R), E, K) \rightsquigarrow \text{Eval}(L, E, \text{KProd}_0 \text{ } K \text{ } R)} \\
\\
\frac{}{\text{Apply}(\text{KProd}_0 \text{ } E \text{ } R \text{ } K, V) \rightsquigarrow \text{Eval}(R, \text{Env}, \text{KProd}_1 \text{ } V \text{ } K)} \\
\\
\frac{}{\text{Apply}(\text{KProd}_1 \text{ } L \text{ } K, V) \rightsquigarrow \text{Apply}(K, \text{VProd } L \text{ } V)} \qquad \frac{}{\text{Eval}(X.0, E, K) \rightsquigarrow \text{Eval}(X, E, \text{KZro } K)} \\
\\
\frac{}{\text{Apply}(\text{KZro } K, \text{VProd } X \text{ } Y) \rightsquigarrow \text{Apply}(K, X)} \qquad \frac{}{\text{Eval}(X.1, E, K) \rightsquigarrow \text{Eval}(X, E, \text{KFst } K)} \\
\\
\frac{}{\text{Apply}(\text{KFst } K, \text{VProd } X \text{ } Y) \rightsquigarrow \text{Apply}(K, Y)} \qquad \frac{}{\text{Eval}(\text{Left } X, E, K) \rightsquigarrow \text{Eval}(X, E, \text{KLeft } K)} \\
\\
\frac{}{\text{Apply}(\text{KLeft } K, V) \rightsquigarrow \text{Apply}(K, \text{VLeft } V)} \qquad \frac{}{\text{Eval}(\text{Right } X, E, K) \rightsquigarrow \text{Eval}(X, E, \text{KRight } K)} \\
\\
\frac{}{\text{Apply}(\text{KRight } K, V) \rightsquigarrow \text{Apply}(K, \text{VRight } V)} \\
\\
\frac{}{\text{Eval}(\text{Case } X \text{ of Left } LN \Rightarrow L \parallel \text{Right } RN \Rightarrow R, E, K) \rightsquigarrow \text{Eval}(X, E, \text{KCase } LN \text{ } L \text{ } RN \text{ } R \text{ } E)} \\
\\
\frac{}{\text{Apply}(\text{KCase } E \text{ } LN \text{ } L \text{ } RN \text{ } R \text{ } K, \text{VLeft } V) \rightsquigarrow \text{Eval}(L, E[LN := V], K)} \\
\\
\frac{}{\text{Apply}(\text{KCase } E \text{ } LN \text{ } L \text{ } RN \text{ } R \text{ } K, \text{VRight } V) \rightsquigarrow \text{Eval}(R, E[RN := V], K)}
\end{array}$$

Fig. 19. Abstract Machine Transition: No Pointer

$$\begin{array}{c}
\frac{}{\text{State}, H \rightsquigarrow \text{State}, H} \qquad \frac{\text{Lookup}(K, H) = KCell}{\text{Eval}(N, E, K), H \rightsquigarrow \text{Apply}(KCell, E(N)), H} \\
\\
\frac{\text{Alloc}(KLet A K C E, H) = (P, H')}{\text{Eval}(\text{Let } A = B \text{ in } C, E, K), H \rightsquigarrow \text{Eval}(B, E, P), H'} \\
\\
\frac{\text{Lookup}(K, H) = KCell \quad \text{Alloc}(\text{Clos } E(\text{fv}) \cdots N C, H) = (P, H')}{\text{Eval}(\backslash N.C, E, K), H \rightsquigarrow \text{Apply}(KCell, P), H'} \\
\\
\frac{\text{Alloc}(KApp_0 K X, H) = (P, H')}{\text{Eval}(F(X), E, K), H \rightsquigarrow \text{Eval}(F, P), H'} \qquad \frac{\text{Alloc}(KProd_0 K R, H) = (P, H')}{\text{Eval}((L, R), E, K), H \rightsquigarrow \text{Eval}(L, E, P), H'} \\
\\
\frac{\text{Alloc}(KZro K, H) = (P, H')}{\text{Eval}(X.0, E, K), H \rightsquigarrow \text{Eval}(X, E, P), H'} \qquad \frac{\text{Alloc}(KFst K, H) = (P, H')}{\text{Eval}(X.1, E, K), H \rightsquigarrow \text{Eval}(X, E, P), H'} \\
\\
\frac{\text{Alloc}(KLeft K, H) = (P, H')}{\text{Eval}(\text{Left } X, E, K), H \rightsquigarrow \text{Eval}(X, E, P), H'} \qquad \frac{\text{Alloc}(KRight K, H) = (P, H')}{\text{Eval}(\text{Right } X, E, K), H \rightsquigarrow \text{Eval}(X, E, P), H'} \\
\\
\frac{\text{Alloc}(KCase LN L RN R E, H) = (P, H')}{\text{Eval}(\text{Case } X \text{ of Left } LN \Rightarrow L \parallel \text{Right } RN \Rightarrow R, E, K), H \rightsquigarrow \text{Eval}(X, E, P), H'}
\end{array}$$

Fig. 20. Abstract Machine Transition: Eval

$$\begin{array}{c}
\frac{\text{Lookup}(K, H) = KCell}{\text{Apply}(KLookup\ K, V), H \rightsquigarrow \text{Apply}(KCell, V), H'} \\
\\
\frac{}{\text{Apply}(KLet\ A\ E\ C\ K, V), H \rightsquigarrow \text{Eval}(C, E[A := V], K), H'} \\
\\
\frac{\text{Lookup}(V, H) = Clos\ E'\ N\ C \quad \text{Alloc}(KApp_1\ E'\ N\ C\ K, H) = (P', H')}{\text{Apply}(KApp_0\ E\ X\ K, V), H \rightsquigarrow \text{Eval}(X, E, P'), H'} \\
\\
\frac{}{\text{Apply}(KApp_1\ E\ N\ C\ K, V), H \rightsquigarrow \text{Eval}(C, E[N := V], K), H'} \\
\\
\frac{\text{Lookup}(K, H) = KCell \quad \text{Alloc}(KProd_1\ V\ E\ K, H) = (P', H')}{\text{Apply}(KProd_0\ E\ R\ K, V), H \rightsquigarrow \text{Apply}(KCell, P'), H'} \\
\\
\frac{\text{Lookup}(K, H) = KCell \quad \text{Alloc}(VProd\ L\ V, H) = (P, H')}{\text{Apply}(KProd_1\ L\ K, V), H \rightsquigarrow \text{Apply}(KCell, P), H'} \\
\\
\frac{\text{Lookup}(V, H) = (VProd\ X\ Y)}{\text{Apply}(KZro\ K, V), H \rightsquigarrow \text{Apply}(KLookup\ K, X), H'} \\
\\
\frac{\text{Lookup}(V, H) = (VProd\ X\ Y)}{\text{Apply}(KFst\ K, V), H \rightsquigarrow \text{Apply}(KLookup\ K, Y), H'} \\
\\
\frac{\text{Lookup}(K, H) = KCell \quad \text{Alloc}(VLeft\ V, H) = (P', H')}{\text{Apply}(KLeft\ K, V), H \rightsquigarrow \text{Apply}(KCell, P'), H'} \\
\\
\frac{\text{Lookup}(K, H) = KCell \quad \text{Alloc}(VRight\ V, H) = (P', H')}{\text{Apply}(KRight\ K, V), H \rightsquigarrow \text{Apply}(KCell, P'), H'} \\
\\
\frac{\text{Lookup}(V, H) = VLeft\ V'}{\text{Apply}(KCase\ E\ LN\ L\ RN\ R\ K, V), H \rightsquigarrow \text{Eval}(L, E[LN := V'], K), H} \\
\\
\frac{\text{Lookup}(V, H) = VRight\ V'}{\text{Apply}(KCase\ E\ LN\ L\ RN\ R\ K, V), H \rightsquigarrow \text{Eval}(R, E[RN := V'], K), H}
\end{array}$$

Fig. 21. Abstract Machine Transition: Apply