Uncomputation

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

Program execution need memory. Program may run out of memory for multiple reasons: big dataset, exploding intermediate state, the machine have less memory than others, etc. When this happens, the program either get killed, or the operating system swaps, significantly degrading the performance. We propose a technique, uncomputation, that allow the program to continue running gracefully even after breaching the memory limit, without significant performance degradation. Uncomputation work by turning computed values back into thunk, and upon re-requesting the thunk, computing and storing them back. A naive implementation of uncomputation will face multiple problems. Among them, the most crucial and the most challenging one is that of breadcrumb. After a value is uncomputed, it's memory can be released but some memory, breadcrumb, is needed, so we can recompute the value back. Ironically, in a applicative language, due to boxing all values are small. This mean uncomputation, implemented naively, will only consume more memory, defeating the purpose. We present a runtime system, implemented as a library, that is absolved of the above breadcrumb problem.

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

ACM Reference Format:
Marisa Kirisame, Heng Zhong, Tiezhi Wang, and Pavel Panchekha. 2018. Uncomputation. *J. ACM* 37, 4, Article 111 (August 2018), 19 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRO

1.1 Motivation

Program execution consume both space and time. While there are tons of research focused on reducing the time spent running a program, memory usage reduction had been consistently underappreciated.

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

- Multi-tenancy. A end-user laptop execute multiple programs concurrently. As an example, a typical programmer might open an IDE to edit and compile code, Zoom for remote meeting, and additionally a browser with dozens of tabs open to lookup information on the internet. Among them, the worst is the web browser, as each tab is it's own separate process, with a renderer, a rule engine, along side 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; 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 time 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 file reaching GBs. Likewise IDEs will parse and analyze the source code of a project, then stored the analyzed results for auto completion. Upon working on huge project such IDEs will crash.
- Intermediate state. A modern computer can read and write memory at a speed of GB per seconds. Without memory reclamation technique such as garbage collection, memory will run out at a matter of seconds. However, some applications is essentially out of reach from garbage collection, and must keep most if not all intermediate states around. These applications include time travelling debugger, jupyter notebook, reverse mode automatic differentiation for deep learning, incremental computation/hash consing, and algorithm that use search, such as model checker and chess bot.
- Low memory limit. Wasm have a memory limit of 2GB, and embedded device or poor people may have device with lower ram. GPU and other accelerator use their own separate memory and usually is of smaller capacity then main memory.

A typical and generic solution to reduce memory consumption is by uncomputation. Uncomputation trade space for time, by tagging values with the metadata that created it(thunk), and when memory is low, deallocate values which might be used later on, recomputing them back with the thunk when the value is needed again.

Another generic solution is swapping, where upon low memory, pages are swapped to disk and swapped back when needed again. While swapping had been implemented for basically all operating systems, it is an especially bad solution for functional and object-oriented languages, as those languages allocate and deallocate lots of small object, yet swapping work on the granularity of pages, unable to separate out the hot objects, that had been recently accessed from the code objects, that had not been accessed.

Typically, uncomputation is implemented in an ad-hoc, case by case basis. When a software consume more memory than desired, it's programmer can (0) look at the program, (1) decide which part is using excessive memory, then (2) add custom data type to record the thunk, alongside the code to replace it (3) implement a cache eviction policy to decide what data to evict(uncompute). Another solution is to analyze the algorithm carefully, replacing the algorithm with a specialized version with recomputation in mind, see gradient checkpointing/iterative deepinging depth first search/island algorithm.

Needless to say, this process is extremely cumbersome, taking precious developers time away from more critical task such as optimizing cpu usage or implementing new features, so a generic, automatic approach is needed to reduce memory consumption.

1.2 Failed Attempt

On first glance, uncomputation is easy, as it's cousin, lazy evaluation, is heavily studied. In lazy computation, object might be represented as thunks, which, when needed, is then computed, and the thunk is replaced by the value. A solution to uncomputation is to modify thunk, so that, upon computing, the old function is still kept. This allow the thunk to uncompute as if it had never been recomputed.

```
Lazy Evaluation: type 'a lazy = MkLazy of ('a, unit -> 'a) either ref

Uncomputation: type 'a uncompute = MkUncompute of 'a option ref * (unit -> 'a)
```

Just like lazy evaluation, we can then uniformally lift all values on top of our modified value type, inserting code that force weak head normal form upon data access, and have some cache policy to decide what to uncompute, which merely rewrite the reference back into the empty optional value.

However, there are multiple problems with this approach:

(1) size measurement. We want to gather statistics to decide what values to evict, and one of the most important statistics is the memory a object consume. However, the heap form a complex object graph, and fast cardinality estimation is highly complex.
(2) dopple ganger. The translation will lift a list into nested uncompute. Now suppose variable X hold a list, while variable Y hold a sublist of X, sharing the same representationbad wording dont know how to fix. When X is uncomputed and recomputed, the corresponding Y part will be duplicated, so there will be two representation of Y in memory. In the extreme case, such duplication can force the program to take exponentially more memory.
(3) bread crumb. A thunk capture other value as free variable, which contain a thunk part, capturing more value. This recursive process capture all value that the current value transitively compute on. Since we still need memory to store the thunk itself, and since a thunk is typically larger then an object, anything we gained on uncomputation, will be offset by the metadata.

While the above three problems seems difficult, we can observe they are all but mere manifestation of recursiveness. In particular, size measurement and dopple ganger deal with recursive objects, and bread crumb deal with recursive thunk. This indicate that a solution that handle recursiveness well naturally solve the three problems at once.

1.3 A recursive Solution

To combat recursiveness we take heavy inspiration from Abstracting Abstract Machine (AAM). AAM lift abstract interpretation onto programming languages with arbitrary feature. The critical problem there is likewise recursiveness, as a naive lifting might allow the lattice infinite merge, causing the analysis to non-terminate. AAM propose to use an abstract machine that abstract over pointers, allocations and lookups to manage recursiveness.

Likewise, the three recursiveness problem listed above can be handled by a similar manner. We started from a purely functional language, specifying it's semantic with the CEK machine. We can then abstract over pointers/allocations/lookups similarly.

Most importantly, our key insight is that by a careful handling of our core data structure, the tock tree, which will be explained later, we can remove a value, alongside the metadata(thunk) on that value, yet still recompute it later. This solve the breadcrumb problem which render the naive apporach unsalvageable.

Alongside our solution is proof that it is both correct and efficient. Our correctness proof state that uncomputing will not effect the final result(preservation), and will not infinite loop(progress). What get computed by the raw, uncompute-free semantic, no matter what we evicted. Our efficiency proof state that if our memory consumption is O(n), where n is the amount of live objects: objects that take O(1) time to force into weak head normal form. In particular, this imply asymptotically we use no space to store any evicted object, yet we somehow can access the metadata that recompute them!

We had implemented our solution, alongside a eviction policy that is both fast to compute, and make better decision over classical eviction policy like LRU, random, and GDSF.

2 OVERVIEW

The tock tree serve as a cache insert this sentence somewhere

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

Section: Implementation (3pg long) Heuristic Loop Unrolling

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

Summarize the meeting into key step

Double O(1)

3 CORE LANGUAGE

Zombie works on a purely functional language with products, sum types, and first-class functions called λ_Z. For simplicity in this paper, we treat the language as untyped. The syntax of λ_Z is shown in Figure 3; its semantics are standard. The full Zombie implementation supports additional features, such as primitive types and input/output; **??** describes how these features are layered on top of the core implementation described here.

Importantly, because Zombie is purely functional, programs are totally deterministic, in the sense that 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 an 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.
(3) Kontinuation, which is to be invoked with the value the Control evaluates to.

In our variation of the CEK machine, we split the evaluation phase from the invocation of the continuation, resulting in a machine state that looks like this:

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

In other words, our CEK machine can be in an Eval or an Apply state; the Eval stores the classic control, environment, and kontinuation, while the Apply state stores just a continuation and a value. The precise syntax of values and kontinuations are given in Figure 5.

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 first sets up an initial state (Eval C {} Done) whose Control is the expression to evaluate, whose Environment is empty, and whose Kontinuation is a special Done continuation. The rules of the CEK machine are then used to transition from one machine state to another, until finally reaching the state Apply Done V. In that state, V is the result of evaluating of C.[1]

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 be something like:

$$\frac{\Gamma \vdash e \rightarrow^* \text{Left } \Gamma' \vdash V}{\Gamma \vdash \text{Case } e\ x\ e_l\ y\ e_r \rightarrow \Gamma', x : V \vdash e_l}$$

[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 Apply Done V.

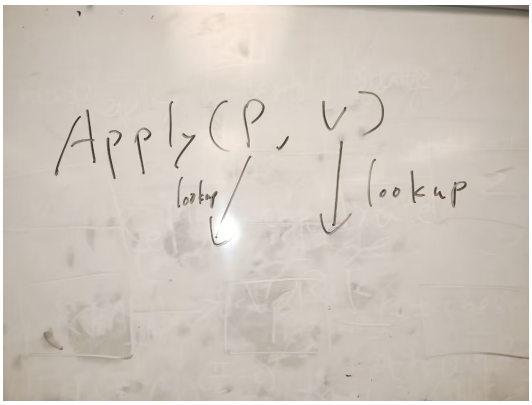

Fig. 1. the machine does a small constant amount of pointer lookup

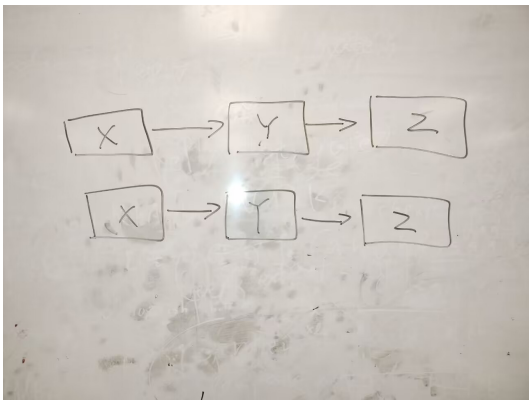

Fig. 2. the deterministic, linear nature of the CEK machine

Here the antecedent of the inference rule might perform arbitrarily many steps, and thus an arbitrary amount of computation; derivations thus form a tree. 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 1.

3.2 Determinism and Tocks

Importantly, the CEK machine is linear and deterministic. This means that 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 the exact same sequence of states in the exact same order. This determinism or "replayability" is essential for Zombie to work, and is illustrated graphically in Figure 2.

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. 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.

Name	N	::=	A set of distinct names
Expr	C	::=	N | Let x = A in B | $\backslash x.\ A$ | $f(x)$ | (x, y) | $x.0$ | $x.1$ | Left x | Right y | Case x of Left $a \Rightarrow L$ || Right $b \Rightarrow R$

Fig. 3. The syntax of λ_Z.

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

Fig. 4. Heap API

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

Fig. 5. Definitions for the CEK Machine

3.3 Heap Memory

Because we are interested the total memory usage of λ_Z program, our variant of the CEK machine includes an explicit heap and explicit pointers. That is, in our formalization values V are formalized as pointers $P\langle \text{VCell} \rangle$ to "value cells". Values cells—which can be closures, products, and sums—in turn contain values, that is, pointers.

During execution, the CEK machine looks up these values using an explicit heap H. To interact with the heap, the CEK machine uses two functions. Lookup : $(P\langle X \rangle, H) \to X$ dereferences a pointer in the heap. Alloc : $(X, H) \to (P\langle X \rangle, H')$ stores a value in the heap, returning a pointer to it and the updated heap. Transitions in the CEK machine call Lookup and Alloc in order to make Eval and Apply steps, as shown in Figure 7 and Figure 8.

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 where it is allocated. This identification is the core abstraction that drives Zombie's implementation.

4 UNCOMPUTING AND RECOMPUTING

merge with next section

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

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

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

$$\frac{}{\mathsf{Eval}(\backslash N.C, Env, K) \rightsquigarrow \mathsf{Apply}(K, \mathsf{Clos}\ Env(\mathrm{fv}) \cdots N\ C)}$$

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 6. Abstract Machine Transition: No Pointer

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

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

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

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

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

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

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

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

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

Fig. 7. Abstract Machine Transition: Eval

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 at 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

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 8. Abstract Machine Transition: Apply

earlier CEK machine states in order to facilitate that. As the program runs, intermediate values can be discarded from the heap to reduce its memory usage.

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

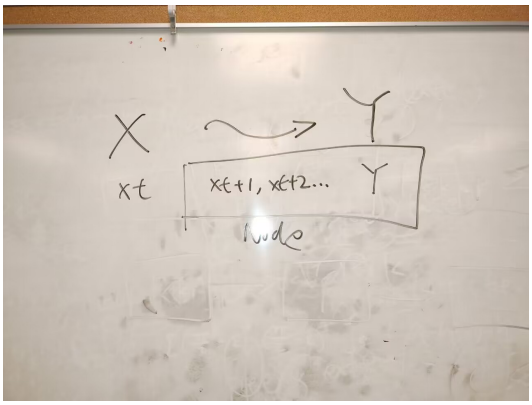

Fig. 9. a node in the tock tree

store asymptotically fewer machine 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 machine state. To recompute an evicted value at some tock i, the tock tree will find the largest machine state at tock $j < i$ and replay from that tock to tock i.

Each node in the Tock Tree stores both a memory cell and a machine state. Specifically, consider the execution of the CEK machine from step t. In step $t + 1$ it allocates a cell; it then performs a computation in step $t + 2$. So the node at tock t contains both the cell allocated at step $t + 1$ as well as the machine state *afterwards*, at step $t + 2$. A transition might not alloc any new cells, in such a case the Node only store a machine state any no memory cell. To be more specific, if the node represent the execution of the CEK machine at step t, it will only contain the CEK machine state at step $t + 1$.

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

4.3 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 correspond to t is found
(2) the original state before replaying, to resume execution at that poin

, 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 called the replay continuation (RK).

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

(1) continuation is a expr with a hole of type expr, while replay continuation is a state with a hole of type cell

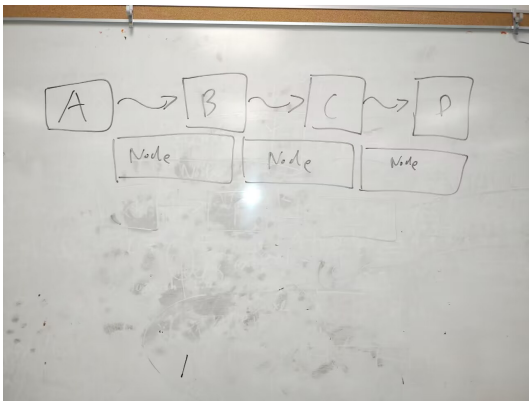

Fig. 10. the tock tree with multiple nodes

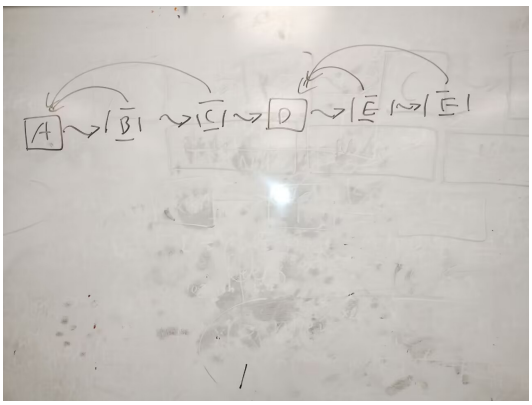

Fig. 11. lookup failure return the latest earlier node

(2) there are way less replay continuation type then continuation: this is because only the state that lookup a cell need a replay continuation; state that does not do not need replay, and so do not need replay continuation.

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

4.4 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 consist 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

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

Fig. 12. Tock Tree API. Note that we deliberately avoid dictating what node get uncomputed, in order to decouple uncomputation/recomputation with selecting what to uncompute. Instead, node might be dropped during insertion into TockTree. One might e.g. set a limit onto the amount of nodes in the Tock Tree.

ReplayContinuation	RK	::=	NoReplay | 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)

(6) The tock tree

And now have 3 modes: in addition to the eval and apply mode, there is a 'return' mode that hold only a cell.

Alongside the 3 modes come 3 kinds of transition:

(1) The happy path: the lookup in the CEK transition rule, is converted into the query transition rule. If the query is successful (the return tock is the queried tock), we enter the happy path, where the transition closely mirror that of the CEK-transition. We collect the cell allocated during transition, alongside the transit-to state, into a node, and insert it into the tock tree. We then transit-to the transit-to state specified by the CEK machine. However, if the tock matching the return continuation is allocated, the machine enter a returning mode instead.
(2) The sad path: on a query that failed(did not get the node matching the tock in return), replay is needed. To replay a value, we push the old state alongside the request tock onto the replay continuation, and transit to the state in the node.
(3) The return path: once a cell matching the top of the replay continuation is found,from the return mode, pop from the replay continuation to resume the state. It will then behave like a normal transition, as we obtained a cell by replaying, and no longer need to query the tock tree. bad wording dont know how to fix

4.5 Reified Continuation

We treat continuations also as values, and label them with tock/put them in the tock tree, just like any other values. This allow us to also uncompute continuation as well. move section alongside tock and cell

In this section we formalize the semantic of the language with uncomputation and recomputation. It is implemented by adding a Replay Continuation alongside the CEK Machine. Hence we call it the CEKR Machine.

$$\frac{}{(\text{State}|\mathit{Return\ Cell}), rk, \mathit{TockTree} \rightsquigarrow (\text{State}|\mathit{Return\ Cell}), rk, \mathit{TockTree}}$$

$$\frac{}{\mathit{CheckReturn}(t, (\text{Nothing}, st), rk) = st} \qquad \frac{t + 1! = t'}{\mathit{CheckReturn}(t, (\_, st), \text{NoReplay}) = st}$$

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

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

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

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

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

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

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

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

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

check return and post process in english everything in figure must be explained. mimic other paper.

Fig. 13. CEKR transition: Return

4.6 Tock Tree

4.7 Replay Continuation

5 IMPLEMENTATION

5.1 Tock Tree

To exploit the temporal/spatial locality, and the 20-80 law of data access (cite?), the tock tree is implemented as a slight modification of a splay tree.

This design grant frequently-accessed data faster access time. Crucially, consecutive insertion take amortized constant time.

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

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

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

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

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

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

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

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

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

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

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

Fig. 14. CEKR Transition: Eval

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

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

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

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

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

$$\frac{Query(tt, V) = (V - 1, (\mathsf{Clos}Env'NC), \_) \qquad st = \mathsf{Eval}(X, Env, t + 1, t + 2) \qquad node = (\mathsf{Just}KApp1Env'NCK'}{\mathsf{Apply}(\mathsf{KApp0}\ Env\ X\ K', V, t), rk, tt \rightsquigarrow PostProcess(t, node, rk, tt)}$$

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

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

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

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

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

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

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

Fig. 15. CEKR Transition: Apply

5.2 Picking Uncomputation Candidate

5.3 Eviction

This allow us to remove any non-leftmost node from the tock tree. After the removal, the query that originally return the removed node, will return the node slightly earlier then that, which we can

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

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

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

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

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

Fig. 16. CEKR Transition: Apply

then replay to regenerate the removed node. In fact, this is the implementation of uncomputation in our system, and any non-leftmost node can be removed, to save memory at any given time. move to implementation section

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

5.3.1 *Union Find.*

5.3.2 *The Policy.*

5.3.3 *GDSF.*

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.

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 return 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 frequent creation of node object, and their insertion to 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 dont think we really need the heap version. I think we should rename the cek with pointer into cekm, and explain it only for introductory purpose (or not at all).

6.1.1 Tock Tree Setup. In order 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 into 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; However, each node is now paired with a boolean bit, indicating whether it had been evicted or not. We will then have a function query_ghost : (TockTree, Tock) -> (Bool, Node) that return the Node, along with a boolean indicating whether the node is present.

6.1.2 CEK/CEKR connection. With query_ghost, we can connect value from the CEKR machine to the CEK machine. More specifically, we can now define a Sem(standing for semantic) function, that convert a CEKR-value to a CEK-value under a given tock tree. The function work by recursively calling query_ghost.

Like wise, we can define equality on continuation, and on state. Additionally, we define 'equality modulo eviction' between two Tock Tree, to be that the two Tock Tree contain the same node, but they might be under different eviction status. Formally speaking, forall Tock, the two Tock Tree must return two equal node, but the eviction bit might differ.

6.1.3 Ghost Stepping. Alongside query_ghost, is the idea of ghost_stepping this name suck. Just like the state transition on CEKR, ghost_stepping is also a transition on State, TockTree tuple. However, unlike the classical state transition, ghost_stepping use query_ghost in place of query, which retrieve the exact node, even if the node is evicted. ghost_stepping is unimplementable, but just like our formalization of Tock Tree and query_ghost, it is purely for formalization purpose. As ghost_step does not need any replay, it does not have the replay continuation. ghost_step serve as a bridge between CEK and CEKR, connecting the two semantic. Once this is completed, we can only talk about correspondence between ghost_step and normal step, inside CEKR, without mentioning the CEK machine at all.

Lemma: CEK stepping is deterministic.

proof: trivial.

Lemma: Ghost stepping is deterministic.

proof: trivial. note that this require the tock tree being deterministic itself. write down this requirement in the right place.

Lemma: Ghost stepping is congruent under eqmodev.

proof: ghost stepping do not read from eviction status.

6.1.4 Well Foundness. A tock tree T is well-founded is this the right word? if:

(1) The left most node exist and is not be evicted. (root-keeping)
(2) Forall node N inserted at tock T, N only refer to tock < T. (tock-ordering)
(3) Forall node pair L R (R come right after L in the tock tree), if (L.state, T) ghost_step to (X, XT), X = R.state and T eqmodev XT. (replay-correct)
(4) Additionally, a (state, tock tree) pair is well-founded if state is in the tock tree. (state-recorded)

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

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

Formally: given CEK-step X, CEKR state Y, if (X, H) = (Y, T) and T is well-founded, (X, H) $\rightsquigarrow$ (X', H'), (Y, T) $\rightsquigarrow$ (Y', T'), (X', H') = (Y', T')

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

The above theorem establish 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 rk is the same as ghost-stepping

proof: trivial

Lemma: if stepping add to rk, the moment it change back, is the same as single ghost-stepping

proof: due to well-foundness the cell RApply recieve must be the same as the same cell on the tock tree but inaccessible. Note that the stepping made here cannot add 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 lexicalgraphical ordering on rk and on state

Theorem: X, rk, tt step-star Y, rk, tt' -> X tt ghost-step-star Y, tt", with tt' eqmodev tt'

proof: the step-star can be decomposed into sequence of no-rk-change step, or a push, with multiple step, 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 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 amount of object with O(1) access cost

7 EVALUATION

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