

Uncomputation

MARISA KIRISAME*, University of Utah, USA

PAVEL PANCHEKHA*, University of Utah, USA

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

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

ACM Reference Format:

Marisa Kirisame and Pavel Panchekha. 2018. Uncomputation. *J. ACM* 37, 4, Article 111 (August 2018), 17 pages. <https://doi.org/XXXXXXX.XXXXXXX>

1 INTRO

1.1 Motivation(1.5pg)

2 INTRO

Program consume memory. Program might use too much memory:

- The input might be huge. Editing a file require loading the whole file into memory; Autocomplete load and parse the whole codebase; Image editors will keep multiple layers of an image, each consuming as much space as the raw image itself.
- Exploding intermediate state: Model checker and Chess bot search through a huge state space, Compiler and Interprocedural analysis expand the AST, Deep Learning/Backpropagation create and keep intermediate value. Additionally, interactive/incrmental application such as Jupyter Notebook, Undo/Redo Stack, Time travelling debugger, program slicing, also keep intermediate state around to revert back to an older state when needed.
- Low memory limit. wasm have a memory limit of 2GB, embedded device or poor people may have device with lower ram. A personal computer might be running multiple programs, which must contend for a fixed memory limit. GPU and other accelerator use their own separate memory and usually is of smaller capacity then main memory.

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

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.

© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.

0004-5411/2018/8-ART111 \$15.00

<https://doi.org/XXXXXXX.XXXXXXX>

Even when there is enough memory, it is still beneficial to use less memory, as multi-tenancy is ubiquitous. Operating System use memory for file cache, the program may have software cache, garbage collector fire less often give more free space, and extra space could be used to increase load or to execute other process.

One common-bought up solution to the memory problem is - to just buy more memory. However this only work to certain extents. Accelerators and wasm have a fixed memory limit and memory-limited algorithm can improve space consumption asymptotically, a feat that merely hardware improvement cannot catch up to. [should i talk about banker paradox here?](#)

However, the only generic technique for reducing memory consumption, swapping, come with great drawback. Swapping move a chunk of data from main memory onto disk, and upon re-requesting it, move the data from disk back into main memory. It degrade performance significantly as disk is far slower then CPU and main memory, and it is especially bad for functional programs as most values is functional programs are boxed, they typically have less cache locality, exacerbated by swapping which operate on pages granularity.

Facing the lack of a generic solution, developers resort to ad-hoc, case-by-case solution, such as cache size tuning, reducing heap size, creating algorithm that swap and uncompute. Such solution complicate the system, and as the developer is likely not an expert on memory-constrained solution, suboptimal design/implementation is often made.

We propose uncomputation, a technique to reduce memory consumption for arbitrary purely functional programs. When needed, uncomputation select some values in the object graph according to cache policy, reverting it back into thunk, releasing memory until it is needed again, in which case it will recompute said values and replacing the thunk with the values. We provided a monadic API behind uncomputation, such that mere mechanical translation is needed to apply our work. The monadic API also grant compatibility, saving developers headache from learning and converting between a full-fledge language.

2.1 Guarantee(0.5pg)

There are multiple guarantees our approach give.

Correctness guarantee: uncomputing will not effect the final result (preservation), and will not be in an infinite loop (progress). Essentially, uncomputing is transparent: the semantic is that of the same as if none happens at all.

Asymptotic guarantee: $O(1)$ access time for all in-memory objects, and $O(1)$ space for each in-memory objects. In particular, this imply that we are not using any space for uncomputed objects.

2.2 Insight

While the above asymptotic guarantee seems very strong, it is actually achievable via:

- Record undelimited context instead of closures. Undelimited context, when run long enough, will produce everything later.
- Assigning a monotonically increasing number, tock, to all zombie creation, and to all call/return/tailcall. This give each zombie a global unique id, so repeat creation of zombie do not create duplicate objects. It also give an ordering so we can find earlier context to replay.
- A data structure, tock tree. The tock tree is a binary search tree, that, on lookup failure, instead of failing (via exception/nullptr/optional), will return the closest match: the value associated with the largest key less then/equal to the input key. This design allow removal of arbitrary context, except the earliest one, as removing an earlier context merely cause lookup to return a further context, requiring more replay.

pavel look here I hate how we talk bad about a naive approach later, but explain our core idea here. I think we should not have naive approach at all, or have it before core idea. Also - 1.3 and 1.4 is new. What other thing do I need for the intro?

3 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 $O(1)$

4 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.

4.1 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) **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.

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:

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

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

¹Note 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 $\text{Apply Done } V$.

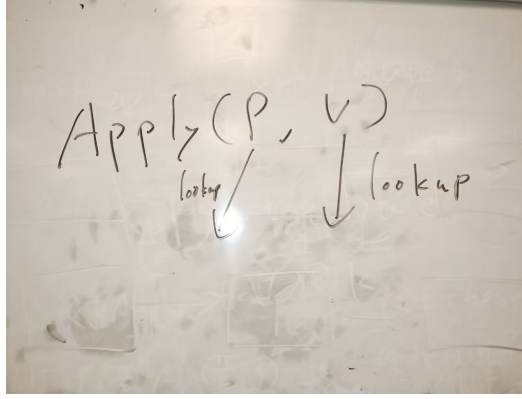


Fig. 1. the machine does a small constant amount of pointer lookup

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:

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

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.

4.2 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.

4.3 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 $P(\text{VCell})$ 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 : $(P(X), H) \rightarrow X$ dereferences a

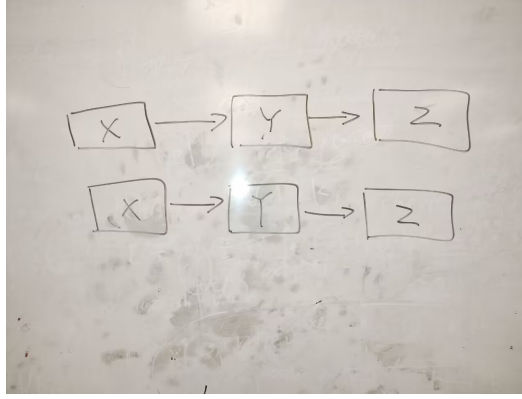


Fig. 2. the deterministic, linear nature of the CEK machine

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

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		$:$	$(\text{Pointer}\langle X \rangle, H) \rightarrow X$
Alloc		$:$	$(X, H) \rightarrow (\text{Pointer}\langle X \rangle, H)$

Fig. 4. Heap API

pointer in the heap. $\text{Alloc} : (X, H) \rightarrow (P\langle X \rangle, H')$ 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.

5 UNCOMPUTING AND RECOMPUTING

5.1 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.

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

Fig. 5. Definitions for the CEK Machine

In our runtime, these “ticks” are implemented as 64-bit integers, though in our model we will treat them as logically unbounded integers. There is a global “current tick” 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 tick i , and that tick 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, ticks are strictly ordered and the value computed at some tick i can only depend on values computed at earlier ticks. Moreover we can recreate the value at tick i by merely rerunning the CEK machine from some earlier state to that point. Because the CEK machine is deterministic, this re-execution will produce the same exact value as the original. Because our pointers refer to abstract values, not to specific memory locations, the heap can thus re-compute a value as necessary instead of storing it in memory.

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

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

5.2 Tock Tree

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

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

Fig. 6. Abstract Machine Transition: No Pointer

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

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

Fig. 7. Abstract Machine Transition: Eval

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.

5.3 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 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 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
- (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.

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

Fig. 8. Abstract Machine Transition: Apply

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.

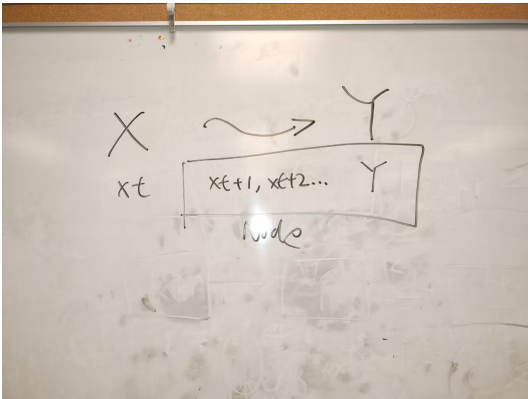


Fig. 9. a node in the tock tree

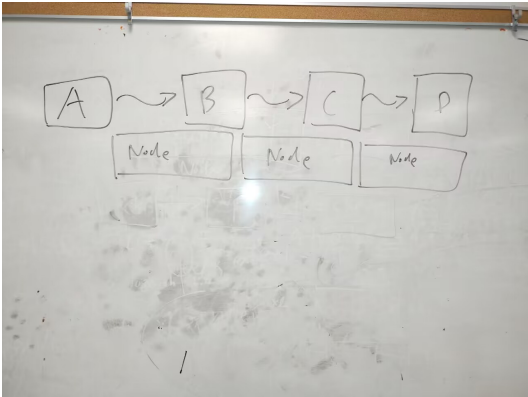


Fig. 10. the tock tree with multiple nodes

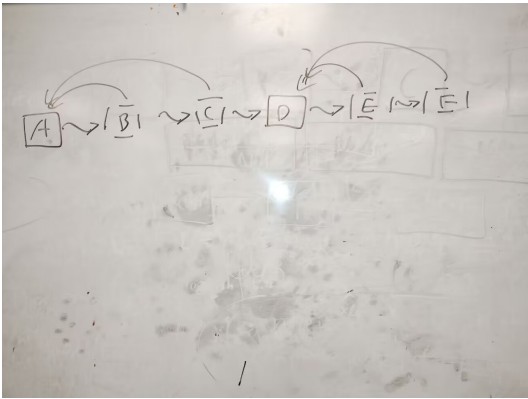


Fig. 11. lookup failure return the latest earlier node

5.4 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) **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

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**

5.5 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.

5.6 Eviction

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

6 CEKR MACHINE

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

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

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	::=	$\text{NoReplay} \mid \text{Replaying Tock RH RK} \mid$
ReplayHole	RH	::=	$\text{RHLookup } V \text{ Tock} \mid$ $\text{RHCASE Env N C N C K Tock} \mid$ $\text{RHZro K Tock} \mid \text{RKFst K Tock} \mid$ $\text{RHApp Env E K Tock}$
Replay	R	::=	$(\text{State}, \text{RK})$

6.1 Tock Tree

6.2 Replay Continuation

7 IMPLEMENTATION

7.1 Tock Tree

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

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

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

7.2 Picking Uncomputation Candidate

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

7.2.1 Union Find.

7.2.2 The Policy.

7.2.3 GDSF.

7.3 Language Implementation

For implementation simplicity and interoperability with other programs, zombie is implemented as a C++ library, and the Cells are ref-counted. Our evaluation compiles the program from the applicative programming language formalized above(give name), to C++ code.

$$\begin{array}{c}
\frac{}{(\text{State}|\text{Return Cell}), rk, \text{TockTree} \rightsquigarrow (\text{State}|\text{Return Cell}), rk, \text{TockTree}} \\
\\
\frac{}{\text{CheckReturn}(t, (\text{Nothing}, st), rk) = st} \qquad \frac{t + 1! = t'}{\text{CheckReturn}(t, (_, st), \text{NoReplay}) = st} \\
\\
\frac{}{\text{CheckReturn}(t, (\text{Just cell}, st), \text{Replaying}(t + 1)rhk) = \text{Return cell}} \\
\\
\frac{t + 1! = t'}{\text{CheckReturn}(t, (\text{Just cell}, st), \text{Replaying}t'rhk) = st} \\
\\
\frac{}{\text{PostProcess}(t, node, rk, tt) = \text{CheckReturn}(t, node, rk), rk, \text{insert}(tt, t, node)} \\
\\
\frac{st = \text{Apply}(cell, v, (t + 1)) \quad node = (\text{Nothing}, st)}{\text{Return cell}, \text{Replaying } _ (RHLookup \ v \ t)rk, tt \rightsquigarrow \text{PostProcess}(t, node, rk, tt)} \\
\\
\frac{st = \text{Eval}(L, \text{Env}[LN := X], K, (t + 1)) \quad node = (\text{Nothing}, st)}{\text{Return } (VLeftX), \text{Replaying } _ (RHCASE \ \text{Env} \ LN \ L \ RN \ R \ K \ t), tt \rightsquigarrow \text{PostProcess}(t, node, rk, tt)} \\
\\
\frac{st = \text{Eval}(R, \text{Env}[RN := Y], K, (t + 1)) \quad node = (\text{Nothing}, st)}{\text{Return } (VRightY), \text{Replaying } _ (RHCASE \ \text{Env} \ 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 } (VProd \ X \ Y), \text{Replaying } _ (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 } (VProd \ X \ Y), \text{Replaying } _ (RHFst \ K \ t), tt \rightsquigarrow \text{PostProcess}(t, node, rk, tt)} \\
\\
\frac{st = \text{Eval}(X, \text{Env}, (t + 1), (t + 2)) \quad node = (\text{Just}(KApp1Env'NCK), st)}{\text{Return } (ClosEnv' \ N \ C), \text{Replaying } _ (RHApp \ \text{Env} \ X \ K \ t), tt \rightsquigarrow \text{PostProcess}(t, node, rk, tt)}
\end{array}$$

Fig. 13. CEKR transition: Return

7.4 Optimization

7.4.1 Fast access path. Querying the tock tree for every value is slow, as it requires multiple pointer traversal. To combat this, each Value is a Tock paired with a weak reference, serving as a cache, to the Cell. When reading the value, if the weak reference is ok, the value is return immediately. Otherwise the default path is executed, and the weak reference is updated to point to the new Result.

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

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

Fig. 14. CEKR Transition: Eval

7.5 Bit counting

8 FORMAL GUARANTEE

8.1 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).

$$\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 Env 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 = (JustKApp1Env'NCK', st)}{Apply(KApp0 Env 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. 15. CEKR Transition: Apply

8.1.1 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

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

Fig. 16. CEKR Transition: Apply

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.

8.1.2 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.

8.1.3 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.

8.1.4 *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 $T \text{ eqmodev } XT$. (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.

8.1.5 *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 R Apply 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: $X, rk, tt \text{ step-star } Y, rk, tt' \rightarrow X \text{ tt ghost-step-star } Y, tt'$, with $tt' \text{ eqmodev } tt$

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.

8.2 Performance

8.2.1 Pebble Game.

8.2.2 Translation to pebble game.

8.2.3 Theorem. memory consumption is linear to amount of object with $O(1)$ access cost

9 EVALUATION

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