Live reload with replay semantics

Frederic Kettelhoit
Recurse Center
fred@fkettelhoit.com

***Abstract*—We present a solution for live reloading that avoids stale state through *replay semantics*: no matter when code is reloaded, the live system state can always be reproduced by replaying the session from the start using only the current code version. Our approach caches side effects while replaying pure computation, ensuring the source code remains the single source of truth. We formalize replay semantics and demonstrate how to implement live reloading with these guarantees in languages that separate pure computation from effects.**

Introduction[1]

Most modern software is developed by writing code, running and testing it, then stopping execution to modify the source files, in a continuous write-run-stop cycle. Minimizing the time it takes to run code can greatly improve development speed, but even with fast compilation, *rebuilding state* during each cycle can significantly impact iteration speed. This is especially true for interactive programs that depend on substantial user input.

Live reloading (sometimes known as *hot reloading*) code is an alternative approach that keeps some or all of the program state alive between modifications. This allows the modified code to reuse previously computed results or input without restarting the program. Existing approaches generally fall into two categories:

- Some programming languages provide support for live reloading on a language level, for example Lisp REPLs [1], Smalltalk images combining state and code [2], and Erlang *hot swapping* [3], [4].
- Some programming languages provide libraries for live reloading in specific situations, such as in game engines [5] or in web development [6]. These approaches limit what is reloaded and when a reload happens (each frame, each time a page is refreshed, etc.).

[1]I thank Florian Ragwitz and David Allen Feil for helpful discussions on this topic at the Recurse Center.

Given that fast feedback cycles are generally desirable when developing software, one might ask why most languages only provide limited support for live reloading in the form of libraries and frameworks, if they provide support for it at all. Why do we not see more adoption of live reloading at the language level? Going further, why are write-run-stop cycles still the norm in software development? Why can't we go from empty source file to fully working software without ever stopping a running program?

Despite the desirability of fast feedback cycles, most languages provide only limited live reloading support. The core problem is *stale state*: when new code versions modify existing functions, the live system may contain data impossible to generate with the current code version. This breaks the fundamental principle that source code should be the single source of truth.

This is the case for many Lisp systems, for example, which often de-emphasize source files in favor of REPL-driven development. While experienced Lisp programmers often have a good mental model of when the current live session may end up containing stale state (and thus require a restart), novice programmers can quickly end up in a state that works in the current session. Such states often break when the program is restarted due to missing function definitions or definitions loaded in the wrong order.

Consider this example where a function increments and prints a counter:

```
let n = 0
function print_n():
    while true:
        n += 1
        print!(n)
```

If we live reload a version that increments by 2 instead of 1, the behavior depends on when reloading occurs. If `n` is odd at reload time, the new version will print odd numbers. The live system could thus end up with

stale state, because an odd number as a value of `n` would not be reachable by restarting the system with the new version of the program.

Traditionally, there have been two approaches to deal with this issue at the language level:

- Stale state is considered to be an acceptable price to pay for live reloading and it is up to the user to restart the system at appropriate times, flushing out the old state. This is the route taken by most systems that support live reloading for development purposes [1], [6]. The burden of deciding when to restart and flush out old state is shifted onto the user.
- State must be explicitly *migrated* whenever a new program version is loaded into the system. In the above example, this might be done by multiplying `n` by 2 or picking the nearest even number. This is the route taken by many systems that support live reloading in production settings, known as *Dynamic Software Updating (DSU)* [7], [8], [9]. Deciding the validity of a migration or deriving a state migration function automatically is undecidable in the general case [10], which limits DSU to specific situations.

Unfortunately, neither of these approaches makes it feasible to incrementally develop a program without ever having to stop the live system. Ignoring stale state leads to defensive restarts, while requiring explicit state migrations adds a considerable burden that is not justified for development purposes where state can often be safely thrown away.

We present an alternative approach, which requires no state migration but nevertheless guarantees that live reloading never leads to a stale state. We show that our approach exhibits *replay semantics*, guaranteeing that no matter when and what code is reloaded, the live state of the system can always be reached by *replaying* the live session from the start, *by using only the most recently loaded version of the code.*

We start by establishing terminology and then give a formal definition of replay semantics. We further present an implementation strategy that leads to a concrete algorithm and finally discuss the limitations of our approach.

Terminology

To ensure clarity, we establish the following terminology used throughout this paper:

- **Effect**: A side-effecting operation that interacts with the outside world, such as I/O operations. We use "effect" and "side effect" interchangeably.
- **Effect signature**: The complete specification of an effect, including both the operation name and its arguments (e.g., `read_num!()` or `print!("hello")`).
- **Effect sequence**: An ordered list of effect signatures observed during program execution.
- **Live reloading**: The process of loading new code into a running system without stopping it.
- **Program version**: A complete snapshot of the source code at a particular point in time. When we say "the new version" we mean the version being loaded during a live reload.
- **Stale state**: Live system state that cannot be reproduced by restarting with the current program version.

Formal definition

We define replay semantics in terms of program executions and their observable effect sequences, noting a similarity to the definition of a *valid state mapping* in [10].

Let a **program** P be a complete source file that can be executed independently. Let an **effect sequence** E be an ordered list of side effects with their arguments, such as `[read_num!(), print!(1), print!(2)]`. We say that two effect signatures are equivalent if they have the same effect name and arguments.

Definition (Replay Semantics): A live reloading system exhibits replay semantics if, for any program P and its reloaded version P', the following property holds:

If executing P' in the live system after reloading results in a final state S, then there exists some effect sequence E such that executing P' from an empty initial state with effect sequence E would also result in state S.

In other words, the live state after any reload is always reachable by restarting the current program version with some appropriate sequence effects. This guarantees that the source code remains the single source of truth, as the live state never depends on code that doesn't exist in the current version.

Intuition: Replay semantics ensure that "time travel" is always possible - you can always achieve the current live state by going back to the beginning and replaying with the new code version, using only the effects that were observed during the live session.

Implementation Strategy

Based on the observation that side effects cannot be replayed but pure computation can, we show how a language implementation can support replay semantics in practice. As a prerequisite, we assume that built-in side effects are syntactically marked in the language using the suffix `!`, such as `print!("hello")` or `read_num!()`. In contrast, regular functions (either user-defined or built-in) are written without a `!` at the end. We do not require a user-defined function that uses side effects to be suffixed with `!`, because it will be evident from its definition (and of the functions it is calling) whether the whole function is pure or not. While our implementation lends itself well to languages with algebraic effects, our approach requires neither an effect system nor effect handlers [11].

As an example, let us return to the program that increments numbers and prints them out in a loop, but change it slightly so that the initial value is dependent on user input:

```
let n = read_num!()
function print_n():
    while true:
        n += 1
        print!(n)

print_n()
```

The key insight leading to replay semantics is extremely simple: A live reloading system will exhibit replay semantics if it *caches side effects* and then *replays* the newly loaded program version using the cached effects. Intuitively, caching side effects (and loading a new version of the pure computation) is equivalent to caching the interaction with the outside world. This means live reloading becomes equivalent to going back in time and restarting the program with the new version, as if the previous version of the pure computation had never existed.

In the above example, loading a new program version that increments `n` by 2 instead of 1 would replay the entire program from the start. But depending on how caching is implemented, the `read_num!()` effect would be cached, guaranteeing that if the user input was an even number, the program will only print out even numbers after loading the new version.

Language support

Before discussing *how and where* to implement caching, let us note a few things. First of all, we are concerned with reloading entire programs. Reloading smaller units (such as modules or functions) is an interesting challenge, but out of scope for the approach discussed here. Second, we can immediately see that by allowing reloading arbitrarily changed programs, a new version could either have side effects that the previous version did not have, or it could omit some side effects that were part of the original version, or rely on the same side effects but in a different order. In other words, there is no guarantee that the new version of a program interacts with the outside world in the same way as the previous version. How caching behaves in the presence of changed *effect sequences* is the central decision that a live reloading system with replay semantics has to make and will be the focus of what follows.

Let us first consider the question of how caching should be exposed in the language. There are 3 options:

1. What is cached is decided by the live system and cannot be controlled by a user from within the language.
2. What is cached and how a cache is built is the responsibility of a user, who can build caching abstractions in the language.
3. What is cached is decided by the user, but caching is provided by the language and is not observable in the absence of reloads.

It would be tempting to hide the decision of what to cache from users (option 1), but knowing which effects can and should be cached depends on how a particular effect interacts with the outside world. In our example, it seems sensible to cache the `read_num!()` effect but to ignore caching for `print!()`. However, there are cases where the same effect might require caching in some programs but not in others.

The other extreme, which would be to leave the *implementation* of caching up to users (option 2), seems appealing from a language design perspective. However, this makes replay semantics impossible, because state in the language could depend on whether something has been cached or even on the size of the cache. This effectively makes older program versions potentially

observable after a reload. Restarting a program would then be different from replaying it after a live reload.

Our approach thus relies on the last option, which strikes a balance between exposing too much and too little to users. We will mark effects that are cached between reloads with a prefix `@` and note that these annotations only play a role during live reloads. Whenever an effect is supposed to be cached *within the execution of a single version*, it needs to be cached by storing the data explicitly in some data structure, with no repercussions for the live reload system.

Cache invalidation

A live reload system that uses caching to guarantee replay semantics needs to choose when and how to invalidate the cache of effects. As an example, let us consider the following program fragment, which uses the `read_num!()` side effect 10 times to wait for user input and caches these side effects between reloads:

```
// v0
let guesses = []
for i in 0..10:
    guesses.push(@read_num!())
```

Let us assume that a new version is loaded into the system, which calls `read_num!()` 9 times instead of 10:

```
// v1
let guesses = []
for i in 0..9:
    guesses.push(@read_num!())
```

Now another version is loaded into the system, which still calls `read_num!()` 9 times as part of the loop, then calls another function `f()` (which may or may not have side effects), then calls `read_num!()` one more time:

```
// v2
let guesses = []
for i in 0..9:
    guesses.push(@read_num!())
f()
guesses.push(@read_num!())
```

Relative to `v0`, the sequence of effects of `v1` is a subsequence that omits one effect, whereas the sequence of effects of `v2` is exactly the same as that of `v0` if and only if `f()` has no effects. In either case, the sequence of `v2` is an extension of `v1`.

Given these relationships, we can now start to describe some of the design decisions that come up in the implementation of a live reload system and ask the following question: When should the cache be invalidated over the course of the above two reloads? In the case of going from `v0` to `v1` the answer seems simple enough. Throughout the whole execution of `v1` the observable effectful behavior is the same as for `v0`, so all of the reads of `v1` can come from the cache.

But what should happen to the last read of `v0`, which is not being used by `v1`? Should this effect be kept in the cache or dropped after the first reload? This is relevant for deciding what happens in the case of `v2` and depending on the situation it could be quite surprising if `v2` suddenly returns the old cached read from `v0`. This brings us to the first decision:

Decision 1: Are unused side effects dropped from the cache after each reload?

Assuming that the last read of `v0` is kept in the cache even after the execution of `v1`, should this read be evicted from the cache if `f()` has side effects? This certainly makes sense if we view the introduction of additional effects as part of `f()` as a *divergence* of `v0` and `v2` in terms of their effect sequences. This is because the effects introduced by `f()` might have an impact on what the last read would be in the absence of caching.

Assuming that the last read of `v0` is dropped from the cache after the execution of `v1`, should the first 9 reads of `v2` return cached versions while the last read bypasses the cache? This might make sense in a case where the reads are mostly independent. However, bypassing the cache only for the last effect could lead to unexpected behavior from a user perspective if each read successively manipulates some state outside of the live system, for example by seeking further into a file. In such a case it might be tempting to invalidate previous effects *retroactively*, so that at the point of the last read of `v2` *all of the reads* of `v2` are invalidated and `v2` is restarted and rerun from scratch. But if `f()` is computationally expensive, this would mean that live reloads could become noticeably less performant than full system restarts. This is because a reload might run the same code more than once. This brings us to the second decision:

Decision 2: Are cached side effects dropped retroactively?

We still need to consider the most fundamental design decision: When are effect sequences considered *divergent*, so that any side effect occurring after the point of divergence is not read from cache? In theory, *any* cache invalidation strategy can guarantee replay semantics, because no matter when and how the cache is invalidated, a reload always replays the pure computation from the start. In practice, however, we might want to provide a stronger guarantee and thus opt for the following invalidation strategy: Whenever the effect sequence of a new version contains an effect that did not occur at the same position in the effect sequence of the previous version, the effect sequences are considered to *diverge* and no further side effects are read from the cache during the execution of the new version.

This gets us closer to an actual implementation, but is still not precise enough. We have not specified which effects we consider to be part of a version's effect sequence: Do we consider only the cached effects or *all* effects (including the ones without a caching annotation)? This brings us to the third decision:

Decision 3: Are uncached side effects relevant for deciding cache invalidation?

We note that different cache invalidation strategies open up a large design space when it comes to replay semantics. An analysis of further cases would be necessary to discuss the different decisions in depth, which is out of scope for the present paper.

Annotation Scoping

In all of the previous examples, effects are directly annotated if they are to be cached. In real programs, however, it will often be necessary to annotate whole functions, which then make use of various effects in their function body.

This can be solved by implementing the `@` annotation as a *combinator* that wraps arbitrary functions and caches all effects that are used as part of the function, as determined by the lexical scope of the function body.

Algorithm

Based on the above discussion, we now present an algorithm for deciding whether to read a side effect from cache or not. Regarding the three decisions mentioned above, we proceed as follows:

1. Unused side effects are dropped from the cache after each reload. This not only improves the efficiency of the implementation by keeping the cache smaller but also leads to a simpler mental model for the user. Cached effects can never *skip* a version, which means that only the previous version needs to be considered.
2. Cached side effects are never dropped retroactively. This guarantees that the live reload system never runs the same version twice and thus maintains the same performance characteristics as a manual restart (apart from some caching overhead). We note that it is possible to explicitly clear the current cache by loading a new version that only differs from the previous one by removing all caching annotations.
3. Uncached side effects are not relevant for deciding cache invalidation. This makes the implementation simpler, because only cached effects need to be considered. It leads to a simpler model where the user only needs to reason about side effects with explicit caching annotations when it comes to reloads.

This leads to the following algorithm, which we present in pseudocode using global variables. We assume that the global variable `cache` starts out empty for the first version and otherwise holds the effect sequence of the previously loaded version.

Each element of the cache is a tuple of the form `(eff_signature, eff_result)`, for example `(print!("hello"), null)` or `(read_num!(), 5)`.

The `cached_effect(i, eff)` function is called once for each effect of the new program version, with `i` being the index of the effect in the sequence of cached effects.

```
let diverged = false
let cache = []

function cached_effect(i, eff):
    if i < cache.length and not diverged:
        let (prev_eff, result) = cache[i]
        if prev_eff == eff:
            return result

    diverged = true
    let result = exec_eff(eff)
    cache.truncate_at(i)
    cache.push((eff, result))
    return result
```

Limitations

While replay semantics provide strong guarantees for live reloading, our approach has several important limitations:

Language Requirements: The approach requires strict separation between pure computation and side effects, with syntactic marking of effects. This may not be practical for existing languages.

Memory Overhead: Caching effect results indefinitely can lead to unbounded memory growth for programs with many unique side effects. Long-running systems may require additional mechanisms for cache management.

Performance Trade-offs: While avoiding full restarts, the replay mechanism still re-executes all pure computation from the beginning on each reload. For computationally intensive programs, this may be slower than traditional live reloading approaches.

Whole-Program Reloading: Our current formalization only addresses reloading complete programs. Extending to module-level or function-level reloading introduces additional complexity around partial state invalidation that we do not address.

Effect Ordering Sensitivity: Programs that rely on precise timing or ordering of effects may behave differently under replay semantics, particularly when some effects are served from cache while others execute fresh.

Despite these limitations, replay semantics offer a principled foundation for live reloading that maintains code as the single source of truth while avoiding the pitfalls of stale state.

References

[1] E. Sandewall, "Programming in an interactive environment: The 'Lisp' experience," *Computing Surveys*, vol. 10, no. 1, pp. 35–71, 1978.

[2] A. Goldberg, *Smalltalk-80: The interactive programming environment*. Reading, MA: Addison-Wesley, 1984.

[3] J. Armstrong, "Making reliable distributed systems in the presence of software errors," PhD thesis, Royal Institute of Technology (KTH), Stockholm, Sweden, 2003. Available: http://erlang.org/download/armstrong_thesis_2003.pdf

[4] J. Armstrong, R. Virding, C. Wikström, and M. Williams, *Concurrent programming in Erlang*, 2nd ed. Prentice Hall, 1996.

[5] Epic Games, "Using live coding to recompile unreal engine applications at runtime." Technical documentation, 2024. Available: https://dev.epicgames.com/documentation/en-us/unreal-engine/using-live-coding-to-recompile-unreal-engine-applications-at-runtime

[6] Webpack Contributors, "Hot module replacement." Webpack Documentation, 2012. Available: https://webpack.js.org/guides/hot-module-replacement/

[7] G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu, "Mutatis mutandis: Safe and predictable dynamic software updating," *ACM Transactions on Programming Languages and Systems*, vol. 29, no. 4, pp. 22:1–22:70, 2007, doi: 10.1145/1255450.1255455.

[8] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol, "Practical dynamic software updating for C," in *Proceedings of the ACM SIGPLAN conference on programming language design and implementation*, in PLDI '06. ACM, 2006, pp. 72–83. doi: 10.1145/1133981.1133991.

[9] H. Seifzadeh, M. Abolhasan, and J. Lipman, "A survey of dynamic software updating," *Journal of Software: Evolution and Process*, vol. 25, no. 5, pp. 535–568, 2013.

[10] D. Gupta, P. Jalote, and G. Barua, "A formal framework for on-line software version change," *IEEE Transactions on Software Engineering*, vol. 22, no. 2, pp. 120–131, 1996.

[11] G. Plotkin and M. Pretnar, "Handlers of algebraic effects," in *Programming languages and systems*, in ESOP 2009, LNCS, vol. 5502. Springer, 2009, pp. 80–94. doi: 10.1007/978-3-642-00590-9_7.