Dead Code Analysis in Reanalyze: Architecture, Algorithm, and Implementation

Codex

November 15, 2025

Abstract

This paper presents a comprehensive description of the dead code analysis performed by the `reanalyze` tool that ships with the ReScript compiler. We describe the inputs, intermediate structures, algorithms, and outputs in sufficient detail that an experienced engineer could reconstruct the analysis. Every major concept is tied directly to the implementation in the repository via inline references.

Contents

1 Introduction

The ReScript compiler ecosystem includes `reanalyze`, a static analysis tool that provides dead code detection, exception analysis, and termination checks. When the dead code elimination (DCE) mode is enabled, `reanalyze` consumes OCaml typed artifacts (`.cmt/.cmti`) and produces warnings, JSON reports, or inline annotations describing unused values, types, exceptions, and modules. The relevant entry point is the CLI defined in `analysis/reanalyze/src/Reanalyze.ml` (function `cli`), which wires command-line switches to the `RunConfig` structure and orchestrates loading of compilation metadata.

This document explains the entire DCE pipeline:

a. Inputs and configuration options and how they propagate through the system.

b. Core data structures that capture declarations, references, and annotations.

c. The algorithm for collection, dependency analysis, liveness resolution, and reporting.

d. The mapping between conceptual stages and actual OCaml modules/files.

e. A detailed walkthrough of the pipeline from invocation to output.

Reference stability. Code links point to the relevant modules (rather than fixed line numbers) because the compiler is under active development; checking the latest revision of the cited files is recommended when cross-referencing behavior.

2 System Overview

2.1 Inputs

Dead code analysis operates on:

- **Typed Artifacts**: `.cmt` and `.cmti` files produced by the compiler for implementation and interface units. The CLI optionally accepts a root directory via `-dce-cmt` (defined in the `speclist` inside `analysis/reanalyze/src/Reanalyze.ml`) to scan for these files.
- **Project Metadata**: The ReScript project root and generated `lib/bs` tree, computed by `processCmtFiles` in `analysis/reanalyze/src/Reanalyze.ml`, provide per-source-directory artifact locations.
- **Configuration Flags**: `analysis/reanalyze/src/RunConfig.ml` defines mutable flags such as `dce`, `transitive`, and suppression lists. Command-line parsing in `analysis/reanalyze/src/Reanalyze.ml` mutates these settings.
- **Annotation Lists**: Users may specify globally live names/paths through `-live-names`, `-live-paths`, and exclusion lists (implemented in `analysis/reanalyze/src/Common.ml`).

2.2 Outputs

Depending on CLI flags, the analyzer emits:

- Human-readable warnings via the generic logging helpers in `analysis/reanalyze/src/Log_.ml`; dead-code specific warnings are constructed in `analysis/reanalyze/src/DeadCommon.ml`.
- Optional JSON diagnostics through `analysis/reanalyze/src/EmitJson.ml`, enabled when the `-json` flag (handled in `analysis/reanalyze/src/Reanalyze.ml`) is set.
- Inline suppression annotations inserted directly into source files when `-write` is supplied; this is implemented in `analysis/reanalyze/src/WriteDeadAnnotations.ml`.
- Statistical summaries for debugging (files analyzed, warnings emitted) via `Log_.Stats` inside `runAnalysisAndReport` (`analysis/reanalyze/src/Reanalyze.ml`); these are independent of JSON emission.

The `-json` and `-write` flags are orthogonal: JSON reports can be generated without touching source files, and annotation writes only occur when explicitly requested.

3 Core Data Structures

3.1 Declaration Registry

All exported declarations are tracked in a positional hash table declared at `analysis/reanalyze/src/DeadCommon.ml:61`. The helper `addDeclaration_` in `analysis/reanalyze/src/DeadCommon.ml:357` inserts entries with:

- Kind (`Value`, `RecordLabel`, `VariantCase`, `Exception`) defined in `analysis/reanalyze/src/Common.ml:166`.
- Module path (list of `Name.t`) describing nested modules.

- Lexical span (`posStart`, `posEnd`) and annotation metadata (`posAdjustment`, `resolvedDead` flag).

Value declarations are added via `analysis/reanalyze/src/DeadCommon.ml:398` (called from `collectValueBinding`); the helper threads the current `ModulePath`, captures side-effect metadata, and stores optional-argument summaries for later diagnostics.

3.2 Reference Graphs

Two positional graphs record references:

- **Value References**: `ValueReferences.table` at `analysis/reanalyze/src/DeadCommon.ml:63` maps target positions to sets of source positions. Edges are added by `addValueReference` (`analysis/reanalyze/src/DeadCommon.ml:91`).
- **Type References**: `TypeReferences.table` (`analysis/reanalyze/src/DeadCommon.ml:71`) records usages of record labels and variant constructors, populated by `DeadType` (Section 5.4).

3.3 File Dependency DAG

`Common.FileReferences` (`analysis/reanalyze/src/Common.ml:55`) stores cross-file edges between source files. The procedure `iterFilesFromRootsToLeaves` (`analysis/reanalyze/src/DeadCommon.ml:110`) topologically orders files based on these edges, ensuring downstream passes respect transitive liveness constraints.

3.4 Annotation Index

The module `ProcessDeadAnnotations` (`analysis/reanalyze/src/DeadCommon.ml:186`) records which positions carry `@dead`, `@live`, or `@gentype` attributes. This informs liveness decisions (Section 4.4) and determines whether to auto-insert new annotations via `WriteDeadAnnotations`.

3.5 Optional Argument Model

`Common.OptionalArgs` defines a structure that tracks call counts, unused optional labels, and always-supplied labels for each function (`analysis/reanalyze/src/Common.ml:129`). `DeadOptionalArgs` builds on this to record references and emit dedicated warnings for unused or redundant optional arguments (`analysis/reanalyze/src/DeadOptionalArgs.ml:84`).

3.6 Module Context Tracking

`ModulePath` (`analysis/reanalyze/src/ModulePath.ml`) maintains the current module stack and source locations while traversing nested modules. Every declaration recorded by `DeadCommon.addDeclaration_` uses `ModulePath.getCurrent` to embed the fully-qualified module path, ensuring that warnings point to the correct namespace even when modules are opened or nested deeply.

3.7 Module-Level Liveness

`DeadModules` (`analysis/reanalyze/src/DeadModules.ml:7`) maintains a map from module names to a boolean indicating whether any live declarations were seen. When all items in a module remain dead, `checkModuleDead` (`analysis/reanalyze/src/DeadModules.ml:22`) reports a module-level warning.

3.8 Suppression Writer

`WriteDeadAnnotations` maintains per-file buffers describing pending `@dead` insertions (`analysis/reanalyze/src/WriteDeadAnnotations.ml:23`). Each buffer line stores the original text plus the declarations to annotate, allowing the tool to emit consistent suppression text for ReScript and OCaml sources alike (`analysis/reanalyze/src/WriteDeadAnnotations.ml:28`).

4 Algorithm

The dead code analysis algorithm comprises five phases (Figure 1).

1. **Collection**: Traverse typed trees to record declarations and direct references.
2. **Reference Enrichment**: Fold compiler-generated dependency metadata and delayed queues (optional args, types, exceptions) into the graphs.
3. **Dependency Ordering**: Build a file-level DAG to determine analysis order.
4. **Liveness Resolution**: Recursively solve for dead declarations with cycle handling and annotation awareness.
5. **Reporting & Annotation**: Emit warnings, JSON entries, module-level diagnostics, and optional inline suppressions.

Figure 1: High-level dead code analysis phases.

4.1 Collection Phase

1. **Load CMT/CMTI**: `loadCmtFile` in `analysis/reanalyze/src/Reanalyze.ml` reads the artifact, resolves its source file, and sets global state (`currentSrc`, `currentModuleName`, etc.).
2. **Process Annotations**: `ProcessDeadAnnotations.signature` or `structure` (defined in `analysis/reanalyze/src/DeadCommon.ml`) is invoked before the main traversal to tag declarations that have `@dead`, `@live`, or `@gentype`. These tags influence every later phase, so the annotation pass must run before `DeadValue` or `DeadType` touch the tree.
3. **Traverse Typed Tree**: For implementation artifacts (`.cmt`), `DeadValue.traverseStructure` (assembled in `analysis/reanalyze/src/DeadValue.ml`) walks expressions, patterns, and structure items. For interfaces (`.cmti`), `DeadCode.processSignature` handles the signature nodes. Each `value_binding` triggers `collectValueBinding`, which registers the declaration and optional arguments via `DeadOptionalArgs.fromTypeExpr`.
4. **Record References**: Expression nodes delegate to `collectExpr` (`analysis/reanalyze/src/DeadValue.ml:114`), which adds edges for identifier uses, optional argument applications, exception constructors, and type field accesses.
5. **Type Collection**: `DeadType.addDeclaration` (`analysis/reanalyze/src/DeadType.ml:82`) registers record labels and variant constructors, while `addTypeReference` (`analysis/reanalyze/src/DeadType.ml:14`) records their usages.

4.2 Reference Enrichment

After traversal, the analyzer processes delayed queues:

- **Value Dependencies**: The compiler-provided `cmt_value_dependencies` list is replayed by `analysis/reanalyze/src/DeadValue.ml:386` to merge aliasing information (e.g., eta-expansion wrappers) via `processValueDependency` (`analysis/reanalyze/src/DeadValue.ml:368`).
- **Optional Arguments**: `DeadOptionalArgs.forceDelayedItems` (`analysis/reanalyze/src/DeadOptionalArgs.ml:64`) updates per-function usage counts.
- **Exceptions**: Deferred exception references are resolved by `analysis/reanalyze/src/DeadException.ml:16`.
- **Types**: `DeadType.TypeDependencies.forceDelayedItems` processes the queued cross-file type dependencies recorded earlier (`analysis/reanalyze/src/DeadType.ml:31`).

4.3 Dependency Ordering

`iterFilesFromRootsToLeaves` (`analysis/reanalyze/src/DeadCommon.ml:110`) builds a priority order of files based on `Common.FileReferences`. Each file's incoming edge count determines when it can be processed; zero-degree files run first, and cycles are optionally reported (`analysis/reanalyze/src/DeadCommon.ml:163`). The resulting order is stored in a hash table and used by `Decl.compareUsingDependencies` (`analysis/reanalyze/src/DeadCommon.ml:446`) to sort declarations prior to resolution.

4.4 Liveness Resolution

For each declaration:

1. Fetch the appropriate reference set (value or type) via `Decl.isValue` (`analysis/reanalyze/src/DeadCommon.ml:436`).
2. Invoke `resolveRecursiveRefs` (`analysis/reanalyze/src/DeadCommon.ml:588`) with an empty set of "currently resolving" positions.
3. During recursion, if a declaration is revisited (cycle), the algorithm assumes it is dead until evidence surfaces, preventing infinite loops.
4. References pointing to annotated positions are filtered out: `declIsDead` (`analysis/reanalyze/src/DeadCommon.ml:578`) removes references to entities marked `@dead` and aborts reporting when `@live` or `@gentype` is present.
5. Once a declaration is definitively dead, it is added to the dead list and `DeadModules.markDead` is notified. Live declarations trigger `DeadModules.markLive` and optional-argument post-processing (`analysis/reanalyze/src/DeadCommon.ml:654`).

4.5 Reporting and Annotation

After resolution:

- Dead declarations are sorted by file/position (`analysis/reanalyze/src/DeadCommon.ml:714`) and fed to `Decl.report` (`analysis/reanalyze/src/DeadCommon.ml:521`), which determines the warning type and message.

- `emitWarning` (`analysis/reanalyze/src/DeadCommon.ml:405`) logs diagnostics, optionally adds inline annotations via `WriteDeadAnnotations.addLineAnnotation` (`analysis/reanalyze/src/WriteDeadAnnotations.ml:139`), and triggers module-level checks.
- `WriteDeadAnnotations.write` flushes buffered edits if `Cli.write` is enabled (`analysis/reanalyze/src/WriteDeadAnnotations.ml:154`).
- `DeadOptionalArgs.check` (`analysis/reanalyze/src/DeadOptionalArgs.ml:84`) emits specific warnings for unused or always-supplied optional parameters on live functions.

The `Suppress` module (`analysis/reanalyze/src/Suppress.ml`) mediates whether a warning can be silenced or annotated at a location, so every call to `emitWarning` first consults `Suppress.filter` before attaching a suggested `@dead`.

5 Implementation Mapping

5.1 CLI and Configuration

`Reanalyze.cli` (`analysis/reanalyze/src/Reanalyze.ml`) parses CLI flags, toggles run modes (`analysis/reanalyze/src/RunConfig.ml:24`), and sets debugging or JSON options (`analysis/reanalyze/src/Common.ml:14`). Each subcommand (`-dce`, `-all`, `-config`) calls helper setters that mutate `RunConfig.runConfig`.

5.2 DeadCode Dispatcher

`DeadCode.processCmt` (`analysis/reanalyze/src/DeadCode.ml`) is the bridge between the CLI and the individual analysis modules: it pattern-matches on `cmt_annots`, runs `ProcessDeadAnnotations`, and then invokes `DeadValue.processStructure` / `processSignature` plus the associated `DeadType` and `DeadOptionalArgs` hooks. In other words, all per-file work originates in `DeadCode`, while the specialized modules focus on their respective AST slices.

5.3 Value Collection

The key routines reside in `DeadValue.ml`:

- `collectValueBinding` (`analysis/reanalyze/src/DeadValue.ml:18`) records declaration metadata, optional arguments, and side-effect analysis.
- `collectExpr` (`analysis/reanalyze/src/DeadValue.ml:114`) adds edges for identifier uses, optional argument applications, field accesses, constructor calls, and custom cases such as lower-case JSX placeholders.
- `traverseStructure` assembles a `Tast_mapper` with custom hooks for expressions, patterns, and structure items (`analysis/reanalyze/src/DeadValue.ml:360`).
- `processValueDependency` integrates compiler-provided alias data to ensure references reflect inlining and eta-expansion artifacts (`analysis/reanalyze/src/DeadValue.ml:368`).

5.4 Type Handling

`DeadType.ml` captures record labels and variant constructors:

- `TypeLabels` caches label locations, enabling cross-file dependency tracking (`analysis/reanalyze/src/DeadType.ml:6`).
- `addDeclaration` (`analysis/reanalyze/src/DeadType.ml:82`) emits declarations for each label/constructor, accounting for inline records and ReScript-specific position adjustments.
- `addTypeDependenciesAcrossFiles` and `addTypeDependenciesInnerModule` (`analysis/reanalyze/src/DeadType.ml:43`) ensure interface/implementation references are linked so that a label defined only in one file is treated correctly.

5.5 Optional Arguments and Exceptions

`DeadOptionalArgs.ml` implements delayed tracking of optional parameter usage (`analysis/reanalyze/src/DeadOptionalArgs.ml:49`). The delay exists because whether an optional parameter is supplied can only be determined while scanning call expressions, yet the function's declaration (and its optional-argument metadata) might be in a different file. The queued data is therefore replayed after every file has been processed, at which point `Common.OptionalArgs` combines the call information with the declaration metadata to decide whether a parameter is unused or always supplied (`analysis/reanalyze/src/Common.ml:129`). `DeadException` handles exceptions defined in one module but referenced elsewhere (`analysis/reanalyze/src/DeadException.ml:9`); ghost-located references are deferred until all files finish processing (`analysis/reanalyze/src/DeadException.ml:16`).

5.6 Central Resolver

`DeadCommon.ml` contains:

- Global tables for declarations and references (`analysis/reanalyze/src/DeadCommon.ml:61`).
- Annotation processing (`analysis/reanalyze/src/DeadCommon.ml:211`).
- File ordering (`analysis/reanalyze/src/DeadCommon.ml:110`).
- Liveness solver (`analysis/reanalyze/src/DeadCommon.ml:588`).
- Reporting helpers (`analysis/reanalyze/src/DeadCommon.ml:521`).

5.7 Output Modules

`WriteDeadAnnotations` manages inline edits (`analysis/reanalyze/src/WriteDeadAnnotations.ml:23`), `EmitJson` produces machine-readable output (`analysis/reanalyze/src/EmitJson.ml:1`), and `Log_` plus `Issues` modules normalize diagnostics for the CLI.

6 Pipeline Walkthrough

Consider running `reanalyze-dce` in a ReScript project:

1. The CLI sets `RunConfig.dce` and resolves the project root (`analysis/reanalyze/src/Reanalyze.ml`).
2. Each source directory under `lib/bs` is enumerated; for every `.cmt/.cmti`, `loadCmtFile` sets `currentSrc` and `currentModuleName`, then triggers `DeadCode.processCmt`.
3. During traversal, value declarations and references funnel into global tables via `addValueDeclaration` and `addValueReference`.
4. After traversal, delayed optional argument, exception, and type dependencies are flushed.
5. Once all files are scanned, `iterFilesFromRootsToLeaves` orders the files, `Decl.compareUsingDependencies` sorts declarations, and `resolveRecursiveRefs` determines deadness.
6. Dead declarations trigger warnings and optional inline annotations; modules with only dead items emit module-level warnings. Optional argument diagnostics run on live functions.
7. If `-write` was set, buffered `@dead` annotations are committed to disk.

7 Module Dependency Overview

```
Reanalyze.cli
    |
    v
DeadCode.processCmt
    |
    +-- ProcessDeadAnnotations
    +-- DeadValue  --> DeadOptionalArgs
    +-- DeadType   --> TypeDependencies
    +-- DeadException
    +-- ModulePath / DeadCommon tables
                |
                v
         DeadCommon.reportDead --> Log_.warning / EmitJson
```

Figure 2: High-level dependency graph between the major analysis modules.

8 End-to-End Pseudocode

The following pseudocode captures the control flow when running with the default configuration (DCE enabled, JSON reporting on, no inline annotation writes). The only runtime inputs are the compiler artifacts (`.cmt/.cmti` files), and the sole output is the JSON file produced by `EmitJson`.

```
(* Default configuration: DCE on, JSON on, no writes *)
let run_default ~project_root =
  RunConfig.dce ();
  Cli.json := true;
  Cli.write := false;
  EmitJson.start ();

  let rec walk dir =
    if Sys.is_directory dir then
      Sys.readdir dir
      |> Array.iter (fun entry -> walk (Filename.concat dir entry))
    else if Filename.check_suffix dir ".cmt"
         || Filename.check_suffix dir ".cmti"
    then load_cmt dir
  and load_cmt path =
    let infos = Cmt_format.read_cmt path in
    match infos.cmt_annots with
    | Implementation structure ->
        let cmti_exists =
          Sys.file_exists (Filename.remove_extension path ^ ".cmti") in
        ProcessDeadAnnotations.structure ~doGenType:(not cmti_exists) structure;
        DeadValue.processStructure
          ~doTypes:true ~doExternals:false
          ~cmt_value_dependencies:infos.cmt_value_dependencies structure;
        DeadType.TypeDependencies.forceDelayedItems ();
        DeadType.TypeDependencies.clear ()
    | Interface signature ->
        ProcessDeadAnnotations.signature signature;
        DeadCode.processSignature ~doValues:true ~doTypes:true signature.sig_type
    | _ -> ()
  in

  walk (Filename.concat project_root "lib/bs");

  DeadException.forceDelayedItems ();
  DeadOptionalArgs.forceDelayedItems ();
  DeadCommon.reportDead ~checkOptionalArg:DeadOptionalArgs.check;
  Log_.Stats.report ();  (* also writes to JSON because Cli.json = true *)
  EmitJson.finish ()
```

This pseudocode now mirrors the production modules: `DeadCode.processCmt` wraps the same operations shown above, the collectors rely on the `DeadValue` and `DeadType` traversals, and JSON output piggybacks on the standard logging pipeline whenever `Cli.json` is enabled.

9 Dead Code Examples

ReScript surface syntax

```
module UserStats = {
  let activeCount = fetchActive ()
  let unusedValue = expensiveComputation()
  let getActive () = activeCount
}

let _ = UserStats.getActive ()
```

The analyzer marks `UserStats.unusedValue` as dead (no references survive traversal) while keeping `activeCount` alive because it feeds `getActive`. If `-write` is enabled, the tool suggests inserting `@dead("UserStats.unusedValue")` next to the declaration.

OCaml-typed tree representation

```
let unused_value =
  let open UserStats in
  let v = expensiveComputation () in
  v
```

After translating the ReScript source to the OCaml typed tree, `unused_value` appears as a standalone `value_binding`. Because no downstream `Texp_ident` references point at its location, `DeadCommon.declIsDead` returns `true` and the value is reported (or auto-annotated) as removable.

10 Reconstruction and Extension

To rebuild or extend the analyzer:

1. **Typed Tree Access**: Implement a walker mirroring `DeadValue.traverseStructure`, ensuring you capture the same metadata (locations, optional args, side effects).
2. **Data Stores**: Recreate the declaration and reference tables (`analysis/reanalyze/src/DeadCommon.ml:61`) along with annotation tracking (`analysis/reanalyze/src/DeadCommon.ml:186`).
3. **Dependency Management**: Mirror `iterFilesFromRootsToLeaves` and `Decl.compareUsingDependencies` to maintain deterministic ordering.
4. **Solver**: Port `resolveRecursiveRefs` and the surrounding helpers to preserve cycle handling, annotation semantics, and module-level reporting.
5. **Diagnostics**: Implement optional argument and exception-specific checks if parity with the existing tooling is required.

11 Testing and Validation

Reanalyze integrates with the broader ReScript test suite, but dedicated validation can follow these steps:

- `maketest-analysis` runs the analysis-specific tests described in `AGENTS.md`.
- Synthetic projects can be placed in `tests/analysis_tests` to ensure new features are covered.
- For regression scenarios, create small `.res/.resi` files, run `reanalyze-dce-write`, and inspect both warnings and inline annotations.

12 Glossary

- **CMT/CMTI**: Compiled module artifacts (`.cmt` for implementations, `.cmti` for interfaces) that contain typed trees plus dependency metadata.
- **Typed tree**: The OCaml `Typedtree` representation produced after type checking; all `DeadVal ue` and `DeadType` traversals operate on this structure.
- **Ghost location**: A `Location.t` whose `loc_ghost` flag is `true`; such positions are ignored for reporting because they do not correspond to concrete source code spans.