Rewatch: Core Algorithms for Incremental Compilation

Abstract

Rewatch is the build and watch engine for ReScript projects. This document presents its core algorithms as typed functions, abstracting away implementation details to expose the fundamental computational patterns. The goal is to identify algorithmic building blocks that could be expressed using incremental and reactive computation primitives.

1 Introduction

A build system for a typed language must solve several interrelated problems:

1. **Discovery**: Find source files and compute a package dependency graph.
2. **Parsing**: Transform source files into abstract syntax trees.
3. **Dependency Analysis**: Extract module-level dependencies from parsed sources.
4. **Compilation**: Compile modules in dependency order, respecting the constraint that a module's interface must be available before its dependents compile.
5. **Incrementality**: When inputs change, recompute only what is necessary.

The key insight is that these problems decompose into a small set of computational patterns: *graph construction*, *fixpoint computation*, *topological scheduling*, and *change propagation*. We present each phase as a pure function with explicit inputs and outputs, then discuss how dirtiness flows through the system.

2 Build Flows Overview

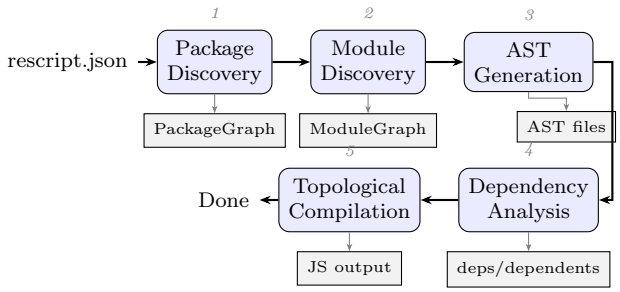
Before diving into the details, we present the two main execution flows: one-shot build and watch mode.

2.1 One-Shot Build

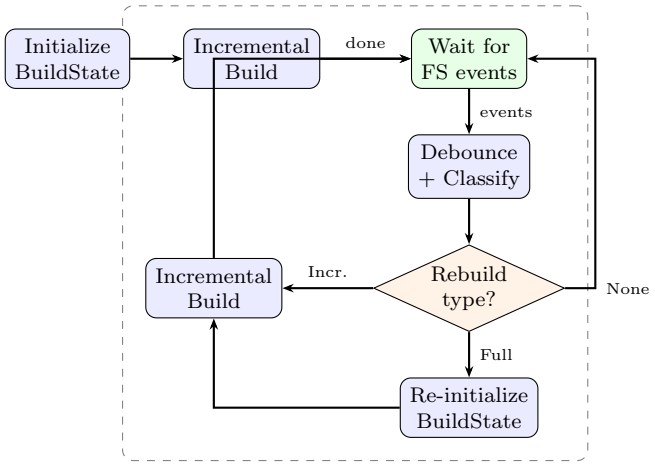
A one-shot build (Figure 1a) executes phases sequentially: package discovery, module discovery, AST generation, dependency analysis, and topological compilation. Each phase produces data consumed by subsequent phases.

2.2 Watch Mode

Watch mode (Figure 1b) wraps the pipeline in an event loop. After an initial build, the system monitors file changes. Content changes mark modules dirty for incremental rebuild; structural changes (file creation/deletion) trigger full re-initialization.



(a) One-shot build: phases 1–3 discover and parse; phases 4–5 analyze and compile.



(b) Watch mode: event loop triggers incremental or full rebuilds based on file changes.

Figure 1: Build flows. (a) One-shot build executes phases sequentially, producing intermediate data (gray boxes). (b) Watch mode wraps the build in an event loop; content changes trigger incremental rebuilds, structural changes trigger full re-initialization.

2.3 Key Data Flow Insight

The central invariant is that the *ModuleGraph* persists across incremental rebuilds in watch mode, with dirty bits controlling which computations are re-executed. This is the essence of incrementality: rather than recomputing everything, we track what changed and propagate dirtiness through the dependency graph.

3 Type Vocabulary

We define a vocabulary of types to describe the state space of the build system. These types abstract over file system paths and other implementation details.

3.1 Identifiers and Paths

We begin with primitive types representing identifiers and content fingerprints:

Path	*Canonical absolute file path*
ModuleName	*Logical module identifier (e.g., "List", "MyApp-Utils")*
PackageName	*Package identifier (from rescript.json)*
Hash	*Content hash for change detection*
Timestamp	*File modification time*

3.2 Source Representation

A source module consists of an implementation file and an optional interface file. Each file carries metadata for tracking changes.

File metadata:

$$\mathsf{FileInfo} = \left\{ \begin{array}{lll} \mathit{path} & : \mathsf{Path} & \\ \mathit{lastModified} & : \mathsf{Timestamp} & \\ \mathit{parseDirty} & : \mathsf{Bool} & \mathit{needs\ re\text{-}parsing?} \\ \mathit{parseState} & : \mathsf{ParseState} & \mathit{Pending} \mid \mathit{Success} \mid \mathit{Warning} \mid \mathit{Error} \end{array} \right\}$$

Source file pair:

$$\mathsf{SourceFile} = \left\{ \begin{array}{ll} \mathit{implementation} & : \mathsf{FileInfo} \\ \mathit{interface} & : \mathsf{Option}\langle\mathsf{FileInfo}\rangle \end{array} \right\}$$

Source type (sum type):

$$\mathsf{SourceType} ::= \mathsf{Source}(\mathsf{SourceFile}) \mid \mathsf{MIMap}$$

where Source represents a regular `.res`/`.resi` pair and MIMap represents a namespace alias file.

3.3 Module Graph

The central data structure is a directed graph of modules with dependency edges.

$$\mathsf{Module} = \left\{ \begin{array}{lll} \mathit{name} & : \mathsf{ModuleName} & \\ \mathit{package} & : \mathsf{PackageName} & \\ \mathit{source} & : \mathsf{SourceType} & \\ \mathit{deps} & : \mathcal{P}(\mathsf{ModuleName}) & \mathit{modules\ this\ depends\ on} \\ \mathit{dependents} & : \mathcal{P}(\mathsf{ModuleName}) & \mathit{modules\ depending\ on\ this} \\ \mathit{compileDirty} & : \mathsf{Bool} & \mathit{needs\ recompilation?} \\ \mathit{depsDirty} & : \mathsf{Bool} & \mathit{dependency\ set\ changed?} \\ \mathit{lastCmiHash} & : \mathsf{Option}\langle\mathsf{Hash}\rangle & \mathit{ABI\ fingerprint} \end{array} \right\}$$

The module graph is a map from names to modules:

$$\mathsf{ModuleGraph} = \mathsf{ModuleName} \rightharpoonup \mathsf{Module}$$

3.4 Package Structure

Packages group modules and declare inter-package dependencies.

$$\mathsf{Package} = \left\{ \begin{array}{lll} \mathit{name} & : \mathsf{PackageName} & \\ \mathit{path} & : \mathsf{Path} & \\ \mathit{namespace} & : \mathsf{Option}\langle\mathsf{String}\rangle & \\ \mathit{deps} & : \mathcal{P}(\mathsf{PackageName}) & \textit{declared dependencies} \\ \mathit{devDeps} & : \mathcal{P}(\mathsf{PackageName}) & \textit{dev-only dependencies} \\ \mathit{sourceFiles} & : \mathsf{Path} \rightharpoonup \mathsf{SourceMeta} & \\ \mathit{modules} & : \mathcal{P}(\mathsf{ModuleName}) & \end{array} \right\}$$

$$\mathsf{PackageGraph} = \mathsf{PackageName} \rightharpoonup \mathsf{Package}$$

3.5 Build State

The complete build state combines packages, modules, and compiler metadata.

$$\mathsf{BuildState} = \left\{ \begin{array}{ll} \mathit{packages} & : \mathsf{PackageGraph} \\ \mathit{modules} & : \mathsf{ModuleGraph} \\ \mathit{moduleNames} & : \mathcal{P}(\mathsf{ModuleName}) \\ \mathit{compilerHash} & : \mathsf{Hash} \\ \mathit{depsInitialized} & : \mathsf{Bool} \end{array} \right\}$$

4 Phase 1: Package Discovery

The first phase discovers packages and constructs the package dependency graph. This is a recursive traversal of the file system following dependency declarations.

4.1 Specification

```
readDependencies : (PackageName, Path) → List<Package>
```

Given a package name and its path, return all transitive dependencies. The algorithm must handle cycles in the dependency graph (which indicate an error).

4.2 Algorithm

```
readDependencies(name, path):
  config = readConfig(path)
  declared = config.dependencies ∪ config.devDependencies
  result = []
  for depName in declared:
    if depName ∉ visited:
      visited.add(depName)
      depPath = resolvePath(depName, path)
      result.append(Package(depName, depPath, readConfig(depPath)))
      result.extend(readDependencies(depName, depPath))
  return result
```

Pattern: This is a *depth-first graph traversal* with memoization to handle shared dependencies. The visited set ensures each package is processed once.

4.3 Package Graph Construction

```
makePackages : Path → PackageGraph

makePackages(root):
  rootPkg = makePackage(readConfig(root), root, isRoot=true)
  deps = flattenDependencies(readDependencies(rootPkg.name, root))
  packages = { rootPkg.name: rootPkg }
  for dep in deps:
    packages[dep.name] = makePackage(dep.config, dep.path, isRoot=false)
  return packages
```

5 Phase 2: Module Discovery

Once packages are known, we enumerate source files and construct the initial module graph.

5.1 Specification

```
discoverModules : PackageGraph → ModuleGraph

discoverModules(packages):
  modules = {}
  for pkg in packages.values():
    for (path, meta) in pkg.sourceFiles:
      moduleName = pathToModuleName(path, pkg.namespace)
      modules[moduleName] = Module {
        name = moduleName,
        package = pkg.name,
        source = makeSourceType(path, meta),
        deps = ∅,
        dependents = ∅,
        compileDirty = true,
        depsDirty = true,
        lastCmiHash = None
      }
  return modules
```

Pattern: This is a *map* operation over packages, flattened into a single module namespace. Namespace handling ensures module names are unique across packages.

6 Phase 3: AST Generation (Parsing)

Parsing transforms source files into ASTs. This phase runs in parallel for independent modules and tracks dirtiness.

6.1 Specification

```
generateAsts : ModuleGraph → (ModuleGraph, List<Error>)
```

For each module whose source is `parseDirty`, invoke the parser. Update parse state and propagate dirtiness to compilation.

6.2 Algorithm

```
generateAsts(modules):
  errors = []
  results = parallel for (name, m) in modules:
    if isParseDirty(m):
      (astPath, err) = parse(m.source.implementation.path)
      (iastPath, ierr) = if m.source.interface then
                           parse(m.source.interface.path)
                         else (None, None)
      yield (name, astPath, iastPath, err ∪ ierr)
    else:
      yield (name, cachedAstPath(m), cachedIastPath(m), ∅)

  for (name, ast, iast, errs) in results:
    modules[name].source.implementation.parseDirty = false
    modules[name].source.interface?.parseDirty = false
    if errs ≠ ∅:
      modules[name].source.implementation.parseState = Error
      errors.extend(errs)
    else:
      modules[name].compileDirty = true  -- propagate to compilation
      modules[name].depsDirty = true     -- deps may have changed

  return (modules, errors)
```

Pattern: This is a *parallel map* with *dirty-tracking*. The key invariant is: if a source file changed, its AST must be regenerated, and downstream phases (dependency analysis, compilation) must be notified.

7 Phase 4: Dependency Discovery

Dependencies are extracted from AST files. Each AST file contains a header listing the modules it imports.

7.1 Specification

```
computeDependencies : ModuleGraph → ModuleGraph
```

For each module with `depsDirty = true`, read its dependency set from the AST. Update both `deps` and `dependents` to maintain the bidirectional graph.

7.2 Algorithm

```
computeDependencies(modules, validModules):
  newDeps = parallel for (name, m) in modules:
```

```
    if m.depsDirty:
      rawDeps = readDepsFromAst(m.astPath)
      deps = normalizeNamespace(rawDeps, m.package.namespace)
      deps = filterValid(deps, validModules, m.package.allowedDeps)
      yield (name, deps)
    else:
      yield (name, m.deps)

  -- Update dependency graph (sequential for consistency)
  for (name, deps) in newDeps:
    oldDeps = modules[name].deps
    modules[name].deps = deps
    modules[name].depsDirty = false

    -- Update dependents for added/removed edges
    for d in (deps \ oldDeps):
      modules[d].dependents.add(name)
    for d in (oldDeps \ deps):
      modules[d].dependents.remove(name)

  return modules
```

Pattern: This is a *graph edge computation* with *incremental update*. The bidirectional edges (deps/dependents) enable efficient traversal in both directions—forward for compilation order, backward for dirty propagation.

8 Phase 5: Incremental Compilation

Compilation is the most complex phase. It must respect the constraint that a module cannot compile until all its dependencies are compiled, and it must propagate changes when a module's interface (ABI) changes.

8.1 The Compile Universe

Not every module needs consideration during compilation—only those that are dirty or transitively depend on dirty modules. We call this the *compile universe*.

```
computeCompileUniverse : ModuleGraph → Set<ModuleName>

computeCompileUniverse(modules):
  dirty = { name | (name, m) ∈ modules, m.compileDirty }
  universe = dirty
  frontier = dirty
  while frontier ≠ ∅:
    newFrontier = ∅
    for name in frontier:
      for dep in modules[name].dependents:
        if dep ∉ universe:
          universe.add(dep)
          newFrontier.add(dep)
    frontier = newFrontier
  return universe
```

Pattern: This is a *forward reachability* computation—a breadth-first traversal along the dependents edges, starting from the dirty set.

8.2 Topological Compilation

Within the compile universe, modules must be compiled in dependency order. Modules with no pending dependencies can compile in parallel.

```
compile : (ModuleGraph, Set<ModuleName>) → (ModuleGraph, List<Error>)

compile(modules, universe):
  compiled = ∅
  errors = []

  -- Ready set: modules whose deps are all outside universe or compiled
  ready = { m ∈ universe | modules[m].deps ∩ universe ⊆ compiled }

  while ready ≠ ∅:
    results = parallel for name in ready:
      m = modules[name]
      if not m.compileDirty:
        yield (name, Ok, unchanged=true)
      else:
        oldHash = computeCmiHash(m)
        err = compileModule(m)
        newHash = computeCmiHash(m)
        yield (name, err, unchanged=(oldHash = newHash))

    for (name, result, unchanged) in results:
      compiled.add(name)
      ready.remove(name)

      if result is Error:
        errors.append(result)
      else:
        modules[name].compileDirty = false

        -- ABI change propagation
        if not unchanged:
          for dep in modules[name].dependents:
            modules[dep].compileDirty = true
            modules[dep].depsDirty = true

      -- Add newly ready modules
      for dep in modules[name].dependents:
        if dep ∈ universe and dep ∉ compiled:
          if modules[dep].deps ∩ universe ⊆ compiled:
            ready.add(dep)

    if ready = ∅ and compiled ≠ universe:
      -- Cycle detected
      cycle = findCycle(universe \ compiled, modules)
      errors.append(CycleError(cycle))
      break

  return (modules, errors)
```

Patterns: This algorithm combines several patterns:

- **Topological scheduling**: Processing nodes level-by-level based on dependency constraints.
- **ABI fingerprinting**: Comparing interface hashes to detect meaningful changes.

- **Backward change propagation**: When an interface changes, dependents are marked dirty for recompilation.
- **Cycle detection**: Identifying strongly connected components that cannot be compiled.

8.3 ABI Change Propagation

The crucial insight for incrementality is that not all recompilations require recompiling dependents. If a module's implementation changes but its interface ('.cmi' file) does not, dependents need not recompile. We track this via content hashing:

```
propagateDirtiness : (ModuleName, Bool, ModuleGraph) → ModuleGraph

propagateDirtiness(name, abiChanged, modules):
  if abiChanged:
    for dep in modules[name].dependents:
      modules[dep].compileDirty = true
      modules[dep].depsDirty = true
  return modules
```

9 Phase 6: Watch Mode and Event Processing

In watch mode, the build system responds to file system changes. Events are batched and debounced before triggering a rebuild.

9.1 Event Classification

File system events are classified by their effect on the build:

$$\mathsf{FsEvent} ::= \mathsf{Modified}(\mathsf{Path}) \mid \mathsf{Created}(\mathsf{Path}) \mid \mathsf{Deleted}(\mathsf{Path}) \mid \mathsf{Renamed}(\mathsf{Path} \times \mathsf{Path})$$

The rebuild decision is a three-valued type:

$$\mathsf{RebuildType} ::= \mathsf{Incremental} \mid \mathsf{Full} \mid \mathsf{None}$$

9.2 Event Processing

```
processEvents : (List<FsEvent>, ModuleGraph) → (ModuleGraph, RebuildType)

processEvents(events, modules):
  rebuildType = None
  for event in events:
    path = eventPath(event)
    if not isRescriptFile(path): continue
    if isInBuildDir(path): continue

    case event of:
      Modified(path):
        moduleName = pathToModule(path, modules)
        modules[moduleName].source.implementation.parseDirty = true
        rebuildType = max(rebuildType, Incremental)

      Created(path) | Deleted(path) | Renamed(_):
        -- Structural change requires full rebuild
```

```
        rebuildType = Full

  return (modules, rebuildType)
```

Pattern: Event processing is a *fold* over events, accumulating both state changes and a rebuild decision. The key distinction is between content changes (incremental) and structural changes (full rebuild).

9.3 Watch Loop

```
watchLoop : (Path, BuildState) → ⊥  -- runs forever

watchLoop(root, state):
  loop:
    events = debounce(drainEventQueue(), 50ms)
    (modules, rebuildType) = processEvents(events, state.modules)
    state.modules = modules

    case rebuildType of:
      None → sleep(50ms)

      Incremental →
        state = incrementalBuild(state)

      Full →
        state = initializeBuild(root)
        state = incrementalBuild(state)
```

10 Summary of Computational Patterns

The algorithms in Rewatch decompose into a small set of reusable patterns:

Pattern	Where Used	Character
Graph traversal (DFS/BFS)	Package discovery, universe expansion	Pure
Parallel map	AST generation, dep extraction	Pure, parallelizable
Incremental graph update	Dependency edge maintenance	Stateful
Topological scheduling	Compilation order	Stateful
Content hashing	ABI change detection	Pure
Dirty bit propagation	Change notification	Stateful
Event folding	Watch mode	Stateful
Debouncing	Event batching	Time-aware

11 Towards Reactive/Incremental Formulation

The patterns identified above suggest natural mappings to incremental computation frameworks:

- **Dependency tracking**: The `deps`/`dependents` bidirectional graph is essentially a dependency graph that incremental frameworks maintain automatically.
- **Dirty propagation**: Marking dependents dirty when a module changes is the core operation of demand-driven incremental computation (self-adjusting computation).

- **ABI fingerprinting**: Comparing old and new values before propagating changes corresponds to "cutoff" in incremental frameworks—avoiding unnecessary downstream recomputation when values are equal.
- **Topological compilation**: The level-by-level processing with ready-set tracking mirrors how incremental frameworks schedule recomputation.
- **Watch loop**: The event-driven rebuild cycle is a reactive pattern—responding to external stimuli (file changes) by triggering computations.

In a future formulation, the explicit dirty bits and manual propagation could be replaced by declarative dependency specifications, with an incremental runtime handling the bookkeeping automatically.

12 Conclusion

Rewatch's algorithms center on maintaining a module dependency graph and efficiently recomputing only what changes. The core operations—graph construction, parallel parsing, dependency extraction, topological compilation, and change propagation—are instances of well-known computational patterns. This decomposition reveals opportunities for expressing the build system using higher-level incremental or reactive abstractions, potentially simplifying the implementation while preserving correctness and performance guarantees.