Haskell Linter/Refactor Tool: Comprehensive Enhancement Plan

Executive Summary

This plan outlines transforming the current `code-conventions` tool from a simple naming convention linter into a powerful, comprehensive Haskell static analysis and refactoring tool that:

1. **Handles Template Haskell** (which weeder cannot)
2. **Detects unused code** across module boundaries
3. **Enforces coding patterns** with configurable rules
4. **Auto-fixes issues** with format preservation
5. **Provides type-aware analysis** using HIE files

Part 1: Current State Analysis

What We Have Now

Parser: `haskell-src-exts` - a third-party parser

- Pros: Simple to use, well-documented
- Cons: Maintenance mode, doesn't support all GHC extensions, parsing bugs

Analysis Capabilities:

- Type signature convention checking (e.g., `LocationId` → `Key Location`)
- Variable naming conventions (e.g., `Entity Page` → `pE`)
- Pattern matching on argument types
- Tracks variable usages in function bodies

Replacement Engine: Line/column-based string replacement

- Pros: Simple
- Cons: Fragile, doesn't preserve formatting, can break on multi-line changes

Configuration: YAML-based with signature and variable rules

Current Limitations

1. **No cross-module analysis** - works file-by-file
2. **No type information** - only syntactic analysis
3. **No Template Haskell support** - `haskell-src-exts` doesn't handle TH well
4. **Fragile replacements** - string-based, not AST-based
5. **Limited pattern matching** - simple text patterns, not AST patterns
6. **No unused code detection** - no dependency graph
7. **Can't handle CPP** - preprocessor directives break parsing

---

Part 2: Research Findings

Why Weeder Can't Handle Template Haskell

Weeder uses HIE files generated by GHC. When Template Haskell runs:

1. TH splices are **expanded at compile time**
2. The expanded code appears in HIE files
3. But the **dependency from the splice to referenced symbols** isn't captured
4. Result: false positives where TH-used code appears "dead"

Example:

```
helperFunction :: String -> String  -- Weeder thinks this is dead
helperFunction = ...

$(generateCode "helperFunction")     -- TH uses helperFunction, but Weeder doesn't
see it
```

Key Insights from Other Tools

Tool	Approach	What We Can Learn
HLint	ghc-lib-parser	Use GHC's actual parser for extension support
Stan	HIE files	Type-aware analysis using compile-time info
Weeder	HIE + graph traversal	Dependency graph for unused code detection
Retrie	Equation-based rules	Safe refactoring with pattern equations
ghc-exactprint	Annotations	Format-preserving transformations
LiquidHaskell	GHC plugin	Compile-time integration for full info

The Template Haskell Solution

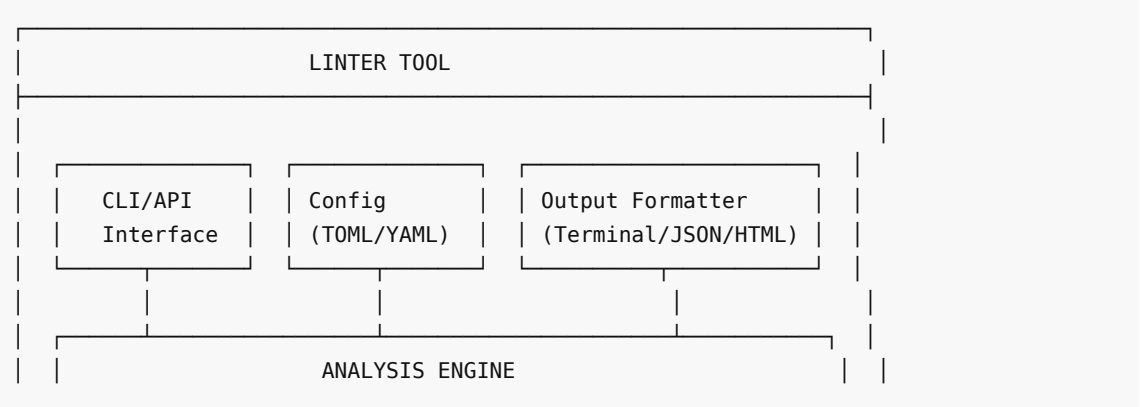
To properly handle Template Haskell, we need **compile-time integration**:

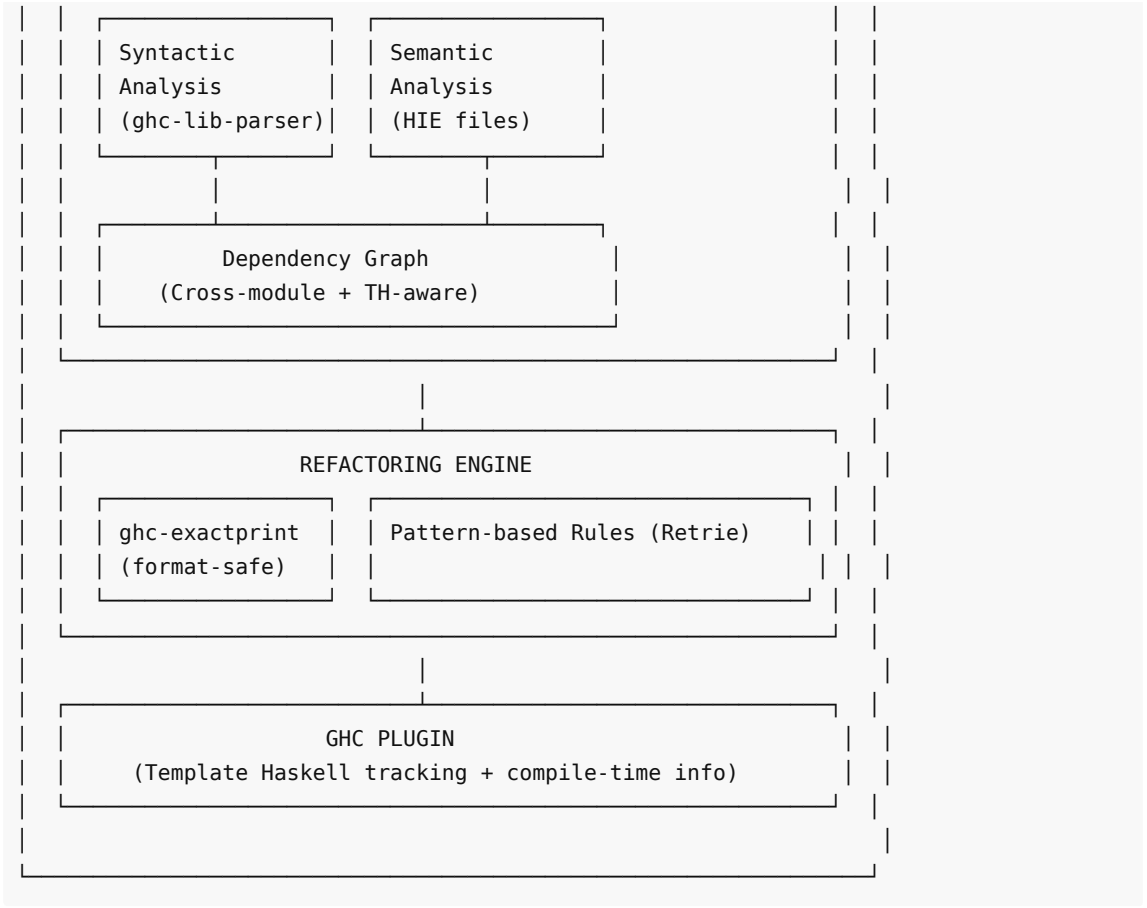
1. **GHC Plugin**: Run as a GHC plugin to see TH expansion results with full context
2. **Source + Spliced Tracking**: Track both pre-expansion and post-expansion references
3. **Name Tracking**: Use GHC's internal Name tracking which survives TH expansion

---

Part 3: Proposed Architecture

Architecture Overview





Three Operational Modes

Mode 1: Quick Analysis (Current-style, Enhanced)

- Uses `ghc-lib-parser` for parsing (replacing `haskell-src-exts`)
- Syntactic-only checks
- Fast, no compilation required
- Good for: CI quick checks, editor integration

Mode 2: Full Analysis (HIE-based)

- Requires compiled project with HIE files
- Type-aware analysis
- Cross-module unused code detection
- Good for: Deep analysis, refactoring large codebases

Mode 3: TH-Aware Analysis (GHC Plugin)

- Runs during compilation as a GHC plugin
- Full Template Haskell support
- Complete dependency tracking
- Good for: Projects with heavy TH usage

Part 4: Feature Specification

4.1 Unused Code Detection

Scope: Detect unused functions, types, imports, and exports across module boundaries

Implementation:

```
1. Parse HIE files → Build symbol table
2. Build dependency graph (symbol → uses-symbol edges)
3. Mark roots (main, exports, test entry points, TH references)
4. Graph reachability analysis
5. Report unreached symbols as unused
```

Template Haskell Handling:

```
-- GHC Plugin approach:
-- 1. Hook into renamer/typechecker
-- 2. Before TH splice: record all Names in scope
-- 3. After TH splice: record all Names referenced in expanded code
-- 4. Build edges: splice → referenced Names
```

Configuration:

```toml
[unused-code]
enabled = true
roots = ["^Main.main$", "^Test.*"]
ignore-modules = ["Generated.*"]
ignore-patterns = ["^_.*"]  # Ignore underscore-prefixed

[template-haskell]
# Names that TH might reference dynamically (can't be statically detected)
dynamic-roots = ["fromLabel", "parseJSON", "toJSON"]
```

4.2 Naming Convention Enforcement (Enhanced)

Current capability (keep):

- Type signature rules: `LocationId` → `Key Location`
- Variable rules: `Entity Page` → `pE`

Enhancements:

- Pattern equations instead of string matching
- Type-aware patterns (using HIE info)
- Scope-aware replacement
- Format preservation

New Configuration Format:

```toml
[[naming.signatures]]
pattern = "* Id"           # Matches "LocationId", "UserId", etc.
replacement = "Key *"      # Becomes "Key Location", "Key User"
message = "Use Key type instead of Id suffix"

[[naming.variables]]
```

```toml
type-pattern = "Entity *"
name-pattern = "*"
replacement = "{type[1]|lowercase|first}E"  # "Entity Page" → "pE"

[[naming.variables]]
type-pattern = "Key *"
name-pattern = "*"
replacement = "{type[1]|lowercase|first}K"  # "Key Location" → "lK"

[[naming.functions]]
pattern = "get* :: IO *"
suggestion = "Consider using MonadIO constraint"
```

4.3 Code Pattern Detection

Purpose: Detect anti-patterns and suggest improvements (like HLint but project-specific)

Rule Types:

```toml
# Anti-pattern detection
[[patterns.warn]]
name = "avoid-head"
match = "head xs"
replacement = "listToMaybe xs"
message = "head is partial, use listToMaybe for safety"

[[patterns.warn]]
name = "avoid-fromJust"
match = "fromJust x"
message = "fromJust is partial, consider pattern matching"

# Project-specific patterns
[[patterns.suggest]]
name = "use-entity-helper"
match = "Entity (entityKey e) (entityVal e)"
replacement = "e"
message = "Redundant Entity reconstruction"

# Type-aware patterns (requires HIE)
[[patterns.warn]]
name = "text-pack-unpack"
match = "pack (unpack x)"
where = "x :: Text"
replacement = "x"
message = "Unnecessary pack/unpack roundtrip"
```

4.4 Import Management

```toml
[imports]
# Detect and remove unused imports
remove-unused = true
```

```
# Detect and suggest missing imports
suggest-missing = true

# Enforce qualified imports for certain modules
qualified-modules = ["Data.Map", "Data.Set", "Data.Text"]

# Enforce explicit import lists
explicit-import-modules = ["*"]  # Or specific modules

# Suggest combining fragmented imports
combine-imports = true
```

4.5 Auto-Fix Engine

Using ghc-exactprint for format preservation:

```
-- Before: fragile string replacement
replaceLintInContent :: Int -> Int -> Text -> Text -> Text -> Text

-- After: AST-based transformation
applyFix :: Fix -> HsModule GhcPs -> Annotated (HsModule GhcPs)
applyFix fix ast = runTransform $ do
  case fix of
    RenameVariable old new -> renameVarInScope old new ast
    ChangeType old new -> changeTypeSignature old new ast
    RemoveUnused name -> removeDeclaration name ast
    ApplyPattern lhs rhs -> applyPatternRule lhs rhs ast
```

Safety features:

- Validate resulting code parses correctly
- Optional: Validate resulting code type-checks (requires compilation)
- Preview mode: show diff before applying
- Undo support: generate reverse patches

Part 5: Implementation Phases

Phase 1: Foundation Upgrade (Weeks 1-3)

Goal: Migrate to ghc-lib-parser, improve replacement engine

Tasks:

1. **Replace haskell-src-exts with ghc-lib-parser**

```
-- Old
import Language.Haskell.Exts

-- New
import GHC.Parser
```

```haskell
import GHC.Driver.Config.Parser
import GHC.Types.SrcLoc
```

2. **Implement ghc-exactprint for replacements**

```
-- Add dependencies
build-depends: ghc-exactprint >= 1.7
             , ghc-lib-parser >= 9.8
```

3. **Port existing rules to new AST format**
   - Rewrite `Extra/Function.hs` for GHC AST
   - Update pattern matching in `Linter.hs`
   - Preserve all existing functionality
4. **Add comprehensive test suite**
   - Property-based tests for round-tripping
   - Regression tests for all current rules

Deliverables:

- Parser upgraded, all tests passing
- Format-preserving replacements working
- No functionality regression

Phase 2: Enhanced Analysis (Weeks 4-6)

Goal: Add HIE file support for type-aware analysis

Tasks:

1. **HIE file parsing**

```haskell
import GHC.Iface.Ext.Binary (readHieFile)
import GHC.Iface.Ext.Types

loadHieFiles :: FilePath -> IO [HieFile]
loadHieFiles dir = do
  paths <- findHieFiles dir
  traverse readHieFile paths
```

2. **Build dependency graph**

```haskell
data DepGraph = DepGraph
  { nodes :: Map Name NodeInfo
  , edges :: Map Name (Set Name)
  , roots :: Set Name
  }

buildDepGraph :: [HieFile] -> DepGraph
findUnused :: DepGraph -> [Name]
```

3. **Type-aware pattern matching**

```
-- Match patterns considering types
matchPattern :: Pattern -> HsExpr GhcTc -> Maybe Match

-- Example: match "pack (unpack x)" only when x :: Text
```

4. **Cross-module analysis**
    - Track exports/imports
    - Build whole-program symbol table
    - Detect unused exports

Deliverables:

- HIE file parsing working
- Basic unused code detection
- Type-aware pattern matching

Phase 3: Template Haskell Support (Weeks 7-10)

Goal: Build GHC plugin for TH-aware analysis

Tasks:

1. **Create GHC plugin skeleton**

```
module LinterPlugin (plugin) where

import GHC.Plugins

plugin :: Plugin
plugin = defaultPlugin
  { renamedResultAction = trackReferences
  , spliceRunAction = trackThSplices
  , pluginRecompile = purePlugin
  }
```

2. **Track TH references**

```
trackThSplices :: [CommandLineOption] -> LHsExpr GhcTc -> TcM (LHsExpr GhcTc)
trackThSplices opts expr = do
  -- Record all Names referenced in the splice
  let refs = collectNames expr
  liftIO $ appendRefs refs
  return expr

-- Collect referenced names from expanded TH
collectNames :: LHsExpr GhcTc -> Set Name
```

3. **Integrate with main analysis**
    - Plugin writes reference data to file

- Main tool reads and incorporates into dep graph
- TH-referenced symbols marked as roots

4. **Handle common TH patterns**
   - Lens generation (`makeLenses`)
   - Aeson instances (`deriveJSON`)
   - Persistent models (`share [mkPersist ...]`)
   - QuasiQuotes

Deliverables:

- Working GHC plugin
- TH references properly tracked
- No false positives for TH-used code

Phase 4: Advanced Refactoring (Weeks 11-14)

Goal: Retrie-style equation-based refactoring

Tasks:

1. **Implement pattern equation parser**

```
-- Rules defined as Haskell equations
data Rule = Rule
  { lhs :: HsExpr GhcPs      -- Pattern to match
  , rhs :: HsExpr GhcPs      -- Replacement
  , conditions :: [Condition] -- Type/scope conditions
  }

parseRule :: Text -> Either Error Rule
parseRule "concat (map f xs) = concatMap f xs" = ...
```

2. **Safe substitution engine**

```
applyRule :: Rule -> HsExpr GhcPs -> Maybe (HsExpr GhcPs)
applyRule rule expr = do
  bindings <- match (lhs rule) expr
  substitute bindings (rhs rule)

-- Handle:
-- - Variable capture avoidance
-- - Parentheses insertion/removal
-- - Operator fixity
```

3. **Batch refactoring mode**

```
# Apply rule across entire codebase
linter refactor --rule "fromJust x = fromMaybe (error msg) x" ./src
```

```
# Interactive mode
linter refactor --interactive ./src
```

4. **Integration with existing rules**
    - Port naming conventions to equation format
    - Combine syntactic and semantic rules

Deliverables:

- Equation-based rule definition
- Safe, format-preserving refactoring
- Batch and interactive modes

Phase 5: Polish & Integration (Weeks 15-16)

Goal: Production-ready tool with excellent UX

Tasks:

1. **Output formats**
    - Terminal with colors and grouping
    - JSON for tooling integration
    - HTML reports (like Stan)
    - SARIF for GitHub Code Scanning
2. **Editor integrations**
    - LSP server mode
    - VSCode extension
    - Editor plugin support
3. **CI/CD support**
    - GitHub Action
    - Exit codes for CI
    - Baseline file for gradual adoption
4. **Documentation**
    - User guide
    - Rule writing guide
    - API documentation

Deliverables:

- Multiple output formats
- CI/CD integration
- Complete documentation

Part 6: Technical Details

6.1 Dependency Changes

Remove:

```
- haskell-src-exts >= 1.23
```

Add:

```
# Core parsing
- ghc-lib-parser >= 9.8
- ghc-lib-parser-ex >= 9.8

# Format preservation
- ghc-exactprint >= 1.7

# HIE file support
- hiedb >= 0.6

# Configuration
- tomland >= 1.3

# Output
- prettyprinter >= 1.7
- prettyprinter-ansi-terminal >= 1.1

# GHC plugin (separate package)
- ghc >= 9.8  # Uses GHC API directly
```

6.2 Module Structure

```
src/
├── Linter/
│   ├── Analysis/
│   │   ├── Syntactic.hs      -- ghc-lib-parser based analysis
│   │   ├── Semantic.hs       -- HIE-based analysis
│   │   ├── DepGraph.hs       -- Dependency graph construction
│   │   └── Unused.hs         -- Unused code detection
│   ├── Rules/
│   │   ├── Naming.hs         -- Naming convention rules
│   │   ├── Patterns.hs       -- Code pattern rules
│   │   ├── Imports.hs        -- Import management
│   │   └── Parser.hs         -- Rule definition parser
│   ├── Refactor/
│   │   ├── Engine.hs         -- ghc-exactprint integration
│   │   ├── Substitution.hs   -- Safe AST substitution
│   │   └── Equations.hs      -- Retrie-style equations
│   ├── Config/
│   │   ├── Types.hs          -- Configuration data types
│   │   ├── Parser.hs         -- TOML parsing
│   │   └── Defaults.hs       -- Default configuration
│   ├── Output/
│   │   ├── Terminal.hs       -- Pretty terminal output
│   │   ├── Json.hs           -- JSON output
│   │   ├── Html.hs           -- HTML reports
```

```
│   │   └── Sarif.hs          -- SARIF format
│   └── Linter.hs             -- Main orchestration
├── Plugin/
│   └── LinterPlugin.hs       -- GHC plugin for TH
└── Main.hs                   -- CLI entry point
```

6.3 Configuration Schema

```toml
# linter.toml

[general]
# Directories to analyze
directories = ["src", "app"]

# Files/directories to exclude
exclude = ["Generated/**", "*.gen.hs"]

# Modes: "quick" (syntactic only), "full" (with HIE), "plugin" (with GHC plugin)
mode = "full"

# Path to HIE files (for "full" mode)
hie-directory = ".hie"

[output]
format = "terminal"  # terminal, json, html, sarif
color = true
group-by = "file"    # file, rule, severity
show-context = true
context-lines = 2

[unused]
enabled = true
check-functions = true
check-types = true
check-imports = true
check-exports = true

# Entry points (always considered used)
roots = [
  "^Main.main$",
  "^.*Spec$",           # Test modules
  "^Paths_.*"           # Cabal generated
]

# Additional roots for TH (can't be detected statically)
template-haskell-roots = [
  "parseJSON",
  "toJSON",
  "makeLenses",
  "share"
]
```

```toml
[naming]
enabled = true

[[naming.types]]
pattern = "(.*)Id"
replacement = "Key {1}"
severity = "error"

[[naming.variables]]
type = "Entity *"
from = "*"
to = "{1|lower|first}E"
severity = "warning"

[[naming.variables]]
type = "Key *"
from = "*"
to = "{1|lower|first}K"
severity = "warning"

[patterns]
enabled = true

[[patterns.rules]]
name = "avoid-head"
severity = "error"
match = "head xs"
fix = "headMay xs"
message = "Use headMay instead of partial head"

[[patterns.rules]]
name = "text-conversion"
severity = "warning"
match = "pack (unpack x)"
where = "x :: Text"
fix = "x"
message = "Unnecessary pack/unpack roundtrip"

[imports]
remove-unused = true
suggest-qualified = ["Data.Map", "Data.Set", "Data.Text", "Data.ByteString"]
require-explicit = true

[fix]
enabled = true
safe-only = true       # Only apply fixes that can't change semantics
preview = true         # Show preview before applying
backup = true          # Create .bak files
```

6.4 CLI Interface

```
# Basic analysis
linter check ./src

# With specific config
linter check --config linter.toml ./src

# Quick mode (syntactic only, no HIE needed)
linter check --mode quick ./src

# Full mode (requires HIE files)
linter check --mode full --hie-dir .hie ./src

# Output formats
linter check --format json ./src > report.json
linter check --format html ./src > report.html

# Fix mode
linter fix ./src                    # Apply all safe fixes
linter fix --interactive ./src      # Prompt for each fix
linter fix --preview ./src          # Show what would be fixed
linter fix --rule avoid-head ./src  # Apply specific rule only

# Unused code detection
linter unused ./src                 # Find unused code
linter unused --roots Main.main ./src

# Import management
linter imports --optimize ./src     # Remove unused, sort, qualify

# Initialize config
linter init                         # Generate default linter.toml

# GHC plugin mode (different binary/invocation)
# Add to ghc-options: -fplugin=LinterPlugin
```

Part 7: Template Haskell Deep Dive

The Core Problem

Template Haskell splices are evaluated at compile time. Consider:

```
-- File: Types.hs
data User = User { userName :: Text, userAge :: Int }

-- File: JSON.hs
import Types
$(deriveJSON defaultOptions ''User)

-- What actually happens:
-- 1. GHC parses $(deriveJSON defaultOptions ''User)
```

```
-- 2. Runs the TH code at compile time
-- 3. Generates instance code that references User, userName, userAge
-- 4. Inserts generated code into the module
-- 5. HIE file shows the generated instance, but not the dependency on User
```

Our Solution: Multi-Layer Tracking

Layer 1: Static Analysis of TH Quotes

```
-- We can statically see that ''User is referenced
findThQuotes :: HsExpr GhcPs -> [Name]
findThQuotes (HsSpliceE _ (HsTypedSplice _ _ _ expr)) =
  findQuotedNames expr  -- Find ''Name and 'name patterns
```

Layer 2: GHC Plugin for Dynamic Tracking

```
-- Track what the TH splice actually generates
spliceRunAction :: [String] -> LHsExpr GhcTc -> TcM (LHsExpr GhcTc)
spliceRunAction _ expr = do
  -- expr is the fully expanded splice result
  let refs = collectAllReferences expr
  recordThDependencies refs
  return expr
```

Layer 3: Known TH Pattern Database

```
# Common TH patterns and what they reference
[[template-haskell.patterns]]
function = "deriveJSON"
references = ["type", "constructors", "fields"]

[[template-haskell.patterns]]
function = "makeLenses"
references = ["type", "fields"]

[[template-haskell.patterns]]
function = "mkPersist"
references = ["type", "fields", "generates-new"]
```

Layer 4: User Annotations

```
-- Allow explicit marking for complex cases
{-# ANN userName ("linter:used-by-th" :: String) #-}
userName :: User -> Text
```

Implementation Strategy

```
data ThDependency = ThDependency
  { thLocation :: SrcSpan      -- Where the splice is
```

```
  , thQuotedNames :: [Name]    -- Names quoted with '' or '
  , thExpandedRefs :: [Name]   -- Names in expanded code (from plugin)
  , thPattern :: Maybe Text    -- Known pattern if matched
  }

mergeThDependencies :: [ThDependency] -> DepGraph -> DepGraph
mergeThDependencies deps graph =
  foldr addThEdges graph deps
  where
    addThEdges ThDependency{..} g =
      -- Add edges from splice to all referenced names
      foldr (addEdge thLocation) g
            (thQuotedNames ++ thExpandedRefs)
```

Part 8: Migration Path

For Existing Users

1. **Phase 1**: Drop-in replacement
   - Same CLI interface
   - Same YAML config format (with TOML as alternative)
   - All existing rules continue to work
2. **Phase 2**: Opt-in new features
   - Add `--mode full` for HIE analysis
   - New TOML config for advanced features
   - Migration guide for config
3. **Phase 3**: Full transition
   - Deprecate old config format
   - New features become default
   - Plugin mode for TH

Example Migration

Before (current config.yaml):

```
signatures:
  - { from: LocationId, to: Key Location }
variables:
  - { type: Entity Page, from: "*", to: pE }
```

After (linter.toml):

```
[[naming.types]]
pattern = "LocationId"
replacement = "Key Location"

[[naming.variables]]
```

```
type = "Entity Page"
from = "*"
to = "pE"
```

Part 9: Success Metrics

Functional Requirements

- [ ] All current test cases pass
- [ ] No false positives for TH-used code
- [ ] Format preservation: `parse . print = id`
- [ ] Cross-module unused detection working
- [ ] < 1 second for quick mode on single file
- [ ] < 30 seconds for full analysis on 10k LOC

Quality Requirements

- [ ] Comprehensive test suite (> 80% coverage)
- [ ] Property-based tests for core functions
- [ ] Integration tests with real codebases
- [ ] Documentation for all public APIs
- [ ] CI/CD integration examples

User Experience

- [ ] Clear, actionable error messages
- [ ] Preview mode for all fixes
- [ ] Gradual adoption path (baseline files)
- [ ] Editor integration working

Part 10: Open Questions & Decisions Needed

Technical Decisions

1. **GHC Version Support**
   - Option A: Support latest GHC only (simpler)
   - Option B: Support GHC 9.4+ (wider adoption)
   - **Recommendation**: Start with latest, add older versions later
2. **Plugin Distribution**
   - Option A: Single package with both tool and plugin
   - Option B: Separate packages (linter + linter-plugin)
   - **Recommendation**: Separate packages for cleaner dependencies
3. **HIE File Generation**
   - Option A: Require user to generate HIE files
   - Option B: Tool generates HIE files on demand

- **Recommendation**: Option A initially, add B as convenience

Feature Prioritization

Must Have (Phase 1-2):

- ghc-lib-parser migration
- Format-preserving replacements
- Existing functionality preserved

Should Have (Phase 3-4):

- HIE-based unused code detection
- Template Haskell support
- Equation-based refactoring

Nice to Have (Phase 5+):

- LSP server
- HTML reports
- GitHub integration

Appendix A: Tool Comparison Matrix

Feature	Current Tool	HLint	Weeder	Stan	Proposed Tool
Parser	haskell-src-exts	ghc-lib	GHC (HIE)	GHC (HIE)	ghc-lib + HIE
Type Info	No	No	Yes	Yes	Yes
Cross-module	No	No	Yes	Yes	Yes
TH Support	No	Limited	No	Limited	Yes (plugin)
Auto-fix	Yes (fragile)	Yes	No	No	Yes (safe)
Custom Rules	Yes (limited)	Yes	No	Limited	Yes (advanced)
Unused Code	No	No	Yes	Partial	Yes
Format Preserve	No	Partial	N/A	N/A	Yes

Appendix B: Reference Implementation Snippets

Parsing with ghc-lib-parser

```haskell
import GHC.Parser
import GHC.Parser.Lexer
import GHC.Data.StringBuffer
import GHC.Types.SrcLoc
import GHC.Driver.Config.Parser
import GHC.Utils.Outputable

parseHaskellFile :: FilePath -> IO (Either String (HsModule GhcPs))
```

```haskell
parseHaskellFile path = do
  content <- readFile path
  let buffer = stringToStringBuffer content
      location = mkRealSrcLoc (mkFastString path) 1 1
      parserOpts = mkParserOpts defaultExtensions [] False False False True
      pState = initParserState parserOpts buffer location
  case unP parseModule pState of
    POk _ (L _ m) -> return (Right m)
    PFailed _ -> return (Left "Parse error")
```

Reading HIE Files

```haskell
import GHC.Iface.Ext.Binary
import GHC.Iface.Ext.Types
import GHC.Types.Name.Cache

loadHieFile :: FilePath -> IO HieFile
loadHieFile path = do
  nameCache <- initNameCache 'z' []
  (hie, _) <- readHieFile nameCache path
  return hie

extractReferences :: HieFile -> [(Name, SrcSpan)]
extractReferences hie =
  [ (name, span)
  | (span, nodeInfo) <- M.toList $ getAsts $ hie_asts hie
  , (Right name, _) <- M.toList $ nodeIdentifiers nodeInfo
  ]
```

GHC Plugin Structure

```haskell
{-# LANGUAGE OverloadedStrings #-}
module LinterPlugin (plugin) where

import GHC.Plugins
import GHC.Tc.Types
import GHC.Hs

plugin :: Plugin
plugin = defaultPlugin
  { renamedResultAction = collectRefs
  , pluginRecompile = purePlugin
  }

collectRefs :: [CommandLineOption]
            -> TcGblEnv
            -> HsGroup GhcRn
            -> TcM (TcGblEnv, HsGroup GhcRn)
collectRefs opts env group = do
```

```
-- Collect all referenced names
let refs = collectNames group
-- Write to output file
liftIO $ appendFile ".linter-refs" (show refs)
return (env, group)
```

Appendix C: Risks and Mitigations

Risk	Impact	Probability	Mitigation
GHC API changes between versions	High	Medium	Use ghc-lib for stability, abstract interfaces
TH patterns we can't track	Medium	Medium	Allow user annotations, pattern database
Performance issues with large codebases	Medium	Medium	Incremental analysis, caching
ghc-exactprint bugs	High	Low	Comprehensive test suite, fallback to string replacement
User adoption	Medium	Medium	Backward compatibility, gradual migration

---

This plan provides a comprehensive roadmap for transforming the current code-conventions tool into a powerful, industry-leading Haskell static analysis and refactoring tool.