# Modern Haskell Tooling

## GHC, Cabal, HLS, and Effect Systems for Professional Development

GHC
CABAL
HLS
EFFECTS


ADRIAN CHIA

# Modern Haskell Tooling

## GHC, Cabal, HLS, and Effect Systems for Professional Development

Steve Publications

This book is available at https://leanpub.com/modernhaskelltooling

This version was published on 2026-09-23

# Contents

# GHC, Cabal, HLS, and Effect Systems for Professional Development

This book is a comprehensive technical reference and practical engineering guide to the modern Haskell ecosystem. It covers the GHC compiler, Cabal build system, Haskell Language Server, and modern effect-system techniques at a depth suitable for professional software engineers building production systems. Through progressively evolving real-world projects, systematic explanations of underlying mechanisms, and rigorous comparison of alternative approaches, you will learn not only how to use these tools but why they work the way they do and how to choose the right architecture for your situation.

# Introduction

In late 2024, a fintech company shipping 2 million lines of Haskell into production reported a particularly frustrating incident: a routine dependency upgrade broke their build in ways that took two days to diagnose [51]. The root cause was not a library bug but a subtle interaction between GHC's specialization optimization, a transitive dependency's upper-bound constraint, and Cabal's solver choosing a version combination that technically satisfied all constraints but failed only at runtime. The incident led the team to rewrite their project's CI pipeline, adopt stricter version bounds, and fundamentally reconsider how they structured their effect system.

This story illustrates why professional Haskell engineering is different from writing Haskell scripts. A script can get away with loose dependencies and ad hoc error handling. A production system that must be maintained by multiple engineers over years requires deliberate choices about tooling, build reproducibility, code organization, and how to handle effects in a way that scales.

Haskell offers world-class tools for this work. GHC is one of the most sophisticated compilers in existence [1], with a runtime system that supports both high-latency-throughput services and low-latency concurrent workloads. Cabal has evolved into a mature build system that manages complex dependency graphs for multi-package projects [11]. The Haskell Language Server brings modern, editor-integrated development to Haskell projects [16]. And the ecosystem's effect systems provide rigorously typed approaches to side effects that can scale from small utilities to large distributed services.

But these tools have their own complexity. GHC's compiler flags, language extensions, and optimization behavior interact in ways that are not documented in any single place. Cabal's solver can produce cryptic error messages when dependencies conflict. HLS requires understanding of how it uses GHC's internals to provide real-time diagnostics. And the choice among MTL-style monad transformers, capability-based effects, and algebraic effect libraries like Polysemy or effectful involves trade-offs in ergonomics, performance, and maintainability that are rarely compared head-to-head in technical depth.

This book aims to fill that gap. It is written for professional software engineers who want to understand and use the modern Haskell ecosystem effectively, whether they are switching to Haskell from another language, evaluating Haskell for production use, or deepening their existing knowledge. I assume you are comfortable with programming concepts, type systems at a basic level, command-line tools, and software engineering practices. I do not assume prior Haskell knowledge.

The book is structured as a coherent progression through the ecosystem. We begin by orienting yourself to modern Haskell and getting your development environment set up. We then dive deep into GHC: how it compiles code, how its type system works in practice, how its optimization passes transform your programs, how its runtime system manages memory and concurrency, how to use GHCi for interactive development, and how to get the most from its diagnostics and warnings. Next, we cover Cabal comprehensively: package structure, dependency resolution, multi-package projects, reproducible builds, and CI integration. After that, we turn to the Haskell Language Server: what it does, how to configure it with your editor, and how its architecture under the hood explains its behavior and limitations.

A substantial portion of the book is devoted to effect systems. We start from the foundations of purity and IO, then move through type-class-based abstraction, MTL-style monad transformers and their real-world architectures, capability-based design patterns, and modern algebraic effects libraries. For each approach we examine not just how it works but how it behaves in practice: its performance characteristics, its debugging story, its composability, and the maintenance burden it imposes on a large codebase. We compare these approaches directly and give concrete recommendations based on project type and size.

Throughout the book we progressively evolve two working projects. The first is a CLI task manager that grows from a minimal program into a properly structured application with tests, benchmarks, profiling, and a well-designed effect architecture. The second is a web service skeleton that demonstrates ReaderT-style and capability-based architectures for production services, including logging, observability, and production configuration. Every code example is complete and runnable: full module declarations, imports, Cabal files, dependencies, language extensions, build commands, and representative output. You can clone these projects, modify them, and use them as reference implementations.

We also dedicate significant space to practical engineering concerns: testing strategies using Hspec, Tasty, QuickCheck, and Hedgehog; profiling and benchmarking to identify and fix performance problems; debugging in a pure functional language; logging and observability for production services; static analysis and code formatting; dependency management at scale; CI pipelines; and systematic approaches to troubleshooting common failure modes like dependency conflicts, confusing type errors, slow compilation, and HLS breakage.

My guiding principle throughout is practical depth over philosophical breadth. Haskell has a rich theoretical foundation, and I respect it. But you are here to build software, not to study category theory. Where theory illuminates practice I will explain it. Where it does not, I will focus on the concrete tools, commands, patterns, and trade-offs that matter in real engineering work. I will be explicit about my assumptions, honest about limitations, and willing to disagree with popular opinions when evidence or experience suggests a different view.

The book targets GHC 9.10 through 9.14, Cabal 3.10 through 3.18, and HLS 2.12 through 2.14. Where behavior differs across versions I will note it explicitly. The ecosystem continues to evolve: GHC is now releasing formal LTS versions starting with 9.14 [10], HLS is adding features at a steady pace [50], and the effects library landscape continues to shift. The concepts and patterns in this book are designed to remain useful as the specific versions and APIs evolve.

Let us begin.

# Chapter 1: The Haskell Ecosystem Today

## Haskell in Production

Haskell is used commercially across aerospace, defense, finance, web applications, hardware design, and biotechnology. This is not a new fact, but the landscape has shifted in recent years toward more mainstream production use rather than isolated research or niche applications.

Major financial institutions including ABN AMRO, Barclays Capital, Deutsche Bank, Standard Chartered, and Alpha Heavy Industries use Haskell for high-frequency trading, derivatives pricing DSLs, and risk analysis [45,46]. The pattern in finance is clear: Haskell's type system and referential transparency make it ideal for domains where correctness is expensive to get wrong and where complex domain logic must be verified against mathematical specifications.

In tech, Facebook uses Haskell for spam detection and internal tools, Google uses it for Ganeti (their cluster management infrastructure), Microsoft uses Bond (a Haskell-based serialization framework), and NVIDIA uses Haskell for internal tooling [45]. Defense and security companies like Galois, BAE Systems, and MITRE use Haskell for cryptography, protocol analysis, and high-assurance systems where formal verification properties matter.

More telling for most readers is the growth of companies where Haskell is a primary production language rather than a curiosity. Mercury, a UK fintech processing payments and lending for hundreds of thousands of businesses, runs a couple million lines of Haskell in production and publishes detailed engineering blog posts about their practices [51]. Hasura, known for its GraphQL engine written in Haskell, ships production services to thousands of customers. IOHK uses Haskell extensively for Cardano blockchain infrastructure.

The Haskell ecosystem supports this production use. Hackage hosts nearly 7,000 packages [41] covering web frameworks (Servant, Yesod, Scotty), async

runtimes (unliftio, async, stm), databases (persistent, postgresql-simple, sqlite-simple), JSON processing (aeson), testing (hspec, tasty, QuickCheck, Hedgehog), logging (fast-logger, co-log), and more. Stackage provides curated, tested snapshots of compatible library versions [43], reducing dependency resolution friction. GHCup makes installing the toolchain reliable across platforms [42].

The reality is that Haskell in production works, but it demands understanding of its toolchain. A production Haskell project is not just Haskell code; it is a carefully configured GHC build with appropriate optimization flags, a Cabal project structure that manages dependencies correctly, an HLS integration that keeps developers productive, and an effect architecture that scales with the codebase. This book exists to make you competent with all of these.

## The Toolchain at a Glance

The modern Haskell toolchain consists of four primary components:

**GHC** (Glasgow Haskell Compiler) is the de facto standard Haskell compiler. It compiles Haskell source code through several intermediate representations (Core, STG, Cmm) down to native machine code or LLVM bitcode [1]. GHC includes its own runtime system (RTS) that manages garbage collection, lightweight threading, and concurrent execution. It provides GHCi, an interactive REPL with debugging capabilities [5]. As of mid-2026, GHC 9.14 is the LTS release, 9.12 is the latest stable, and 9.10 remains widely used in production [44]. GHC follows a formal LTS policy starting with 9.14: LTS versions receive at least two years of bugfix support with six months of overlap with the next LTS [10].

**Cabal** is the package system and build tool for Haskell. It manages package descriptions (.cabal files), dependency resolution, compilation, testing, and installation [11]. Cabal communicates with Hackage, the central package repository. Modern Cabal (since 2.0) uses a "Nix-style local build" system where projects are configured via cabal.project files and builds are cached in dist-newstyle [12]. Cabal handles multi-package projects, local dependencies, solver constraints, and reproducible builds through cabal.freeze files.

**HLS** (Haskell Language Server) implements the Language Server Protocol for Haskell, providing editor-integrated features including real-time diagnostics, code completion, navigation, refactoring, formatting, and expression evaluation [16]. HLS works by running GHC's type checker in the background

on your source files. It supports projects built with Cabal, Stack, and other tools. HLS is versioned separately from GHC but tracks it closely: HLS 2.14 supports GHC versions from 9.6 through 9.14 [18].

**GHCup** (Glasgow Haskell Compiler Upgrade Utility) is the recommended installer for the Haskell toolchain. It manages GHC, Cabal, HLS, and other tools across multiple versions, making it easy to switch between GHC versions for different projects [42].

These tools interact tightly. HLS depends on GHC to type-check your code. Cabal drives GHC during compilation. The versions you choose for each must be compatible. A Cabal project's configuration determines which GHC flags are passed. HLS reads that configuration to provide accurate diagnostics.

Understanding each tool individually is necessary but insufficient. You also need to understand their interactions: how Cabal's solver choices affect which GHC version you use, how GHC's language extensions affect what HLS can parse, how GHC's optimization behavior affects your performance profiling results, and how your project structure affects all of the above.

## Getting Started: Installing GHC and Cabal

Install the Haskell toolchain using GHCup. This is the modern, cross-platform installer that manages GHC, Cabal, HLS, and other tools.

On macOS, Linux, or Windows, run:

```
curl --proto '=https' --tlsv1.2 -sSf https://get-ghcup.haskell.org | sh
```

On Windows PowerShell:

```
Invoke-RestMethod -Uri https://get-ghcup.haskell.org -OutFile ghcup.ps1;
↪  powershell.exe -ExecutionPolicy Bypass -File ./ghcup.ps1
```

After installation, GHCup prompts you to install a GHC version. Choose the LTS release (9.14 as of this writing) or a recent stable version (9.12 or 9.10). GHCup automatically installs the matching Cabal version alongside GHC.

Verify your installation:

```
$ ghc --version
The Glorious Glasgow Haskell Compilation System, version 9.14.1

$ cabal --version
cabal-install version 3.18.1.0
compiled using version 3.18.1.0 of the Cabal library
```

Install HLS through GHCup:

```
$ ghcup install hls
[ Info  ] hls installation successful
$ haskell-language-server-wrapper --version
haskell-language-server version: 2.14.0.0 (GHC: 9.10.3) (PATH:
↪  /home/user/.ghcup/hls/2.14.0.0/lib/haskell-language-server-2.14.0.0/bin/h
↪  askell-language-server-wrapper)
```

The wrapper executable automatically selects the correct HLS binary for your project's GHC version.

Configure your Cabal global settings if needed. The file ~/.cabal/config (or equivalent on Windows) controls default behavior. For a production-oriented setup, consider:

```
jobs: 8
install-method: copy
with-compiler: ghc-9.14.1
```

The jobs setting enables parallel compilation. install-method: copy avoids symlinks on some platforms. with-compiler pins the GHC version.

## A Minimal Working Project

Let us create a minimal, complete Haskell project to establish a baseline. We will progressively evolve this project throughout the book as we introduce new tools and patterns.

Create a project directory:

```
$ mkdir task-manager && cd task-manager
```

Initialize with Cabal:

```
$ cabal init --cabal-version="3.0" --license=MIT --author="Your Name"
```

Cabal prompts interactively. Choose: executable application, name it task-manager, and accept default settings. This creates:

```
task-manager/
  task-manager.cabal
  CHANGELOG.md
  LICENSE
  README.md
  app/
    Main.hs
```

Examine task-manager.cabal:

```
cabal-version: 3.0
name:                task-manager
version:             0.1.0.0
license:             MIT
author:              Your Name
maintainer:          your.email@example.com
build-type:          Simple

executable task-manager
  main-is:           Main.hs
  other-modules:
  build-depends:     base >=4.14 && <5.3
  hs-source-dirs:    app
```

This file declares the package metadata and one executable component. The build-depends line specifies dependencies with version constraints. base is the standard library, and the constraint >=4.14 && <5.3 means this package works with base versions from 4.14 up to but not including 5.3.

Open app/Main.hs and replace the template with:

```haskell
module Main (main) where

main :: IO ()
main = do
    putStrLn "Task Manager v0.1.0"
    putStrLn "Usage: task-manager [list|add|complete] [args...]"
    putStrLn "No arguments: print this message"
```

Build the project:

```
$ cabal build
Resolving dependencies...
Build profile: -w ghc-9.14.1 -O1
In order, the following will be built (use -v for more details):
 - task-manager-0.1.0.0 (exe:task-manager)
[1 of 1] Compiling Main             ( app/Main.hs,
↪  dist-newstyle/build/x86_64-linux/ghc-9.14.1/task-manager-0.1.0.0/x/task-m⌋
↪  anager/build/task-manager/task-manager-tmp/Main.o
↪  )
Linking dist-newstyle/build/x86_64-linux/ghc-9.14.1/task-manager-0.1.0.0/x/ta⌋
↪  sk-manager/build/task-manager/task-manager

$ cabal run task-manager
Task Manager v0.1.0
Usage: task-manager [list|add|complete] [args...]
No arguments: print this message
```

Cabal resolved dependencies, compiled the single module at optimization level 1 (default), and linked the executable in dist-newstyle. The cabal run command finds and executes the built binary.

Note the structure. cabal-version: 3.0 names the .cabal file format version to use (a newer Cabal reads it and accepts newer versions as well). build-type: Simple means use the standard build process from Setup.hs. The executable stanza declares an executable component with its entry point, dependencies, and source directory.

## The Development Loop

Professional Haskell development follows a loop of editing, building, running, testing, and iterating. Let us establish the basic workflow.

First, create a cabal.project file to configure the project. This file controls solver behavior, package selection, and build options for the entire project:

```
$ cat > cabal.project << 'EOF'
packages: .

with-compiler: ghc-9.14.1

flags: +development

-- Allow cabal to use newer versions when needed, but warn
allow-newer: True
EOF
```

Update task-manager.cabal to use a flag for development vs production:

```
cabal-version: 3.0
name:                task-manager
version:             0.1.0.0
license:             MIT
author:              Your Name
maintainer:          your.email@example.com
build-type:          Simple

flag development
  description: Enable development-time options
  default:       False
  manual:        True

executable task-manager
  main-is:           Main.hs
  build-depends:     base >=4.14 && <5.3
                   , text
  hs-source-dirs:    app
  default-language:  GHC2024
  ghc-options:
    -Wall
    -Wcompat
    -Wredundant-constraints
  if flag(development)
    ghc-options: -debug -threaded
```

We added text as a dependency, set GHC2024 as the default language edition, enabled warnings with -Wall -Wcompat -Wredundant-constraints, and added a conditional compilation flag for development builds that enables -debug and -threaded.

Build again:

```
$ cabal build
Resolving dependencies...
Build profile: -w ghc-9.14.1 -O1
In order, the following will be built (use -v for more details):
 - task-manager-0.1.0.0 (exe:task-manager) (configuration changed)
Configuring executable 'task-manager' for task-manager-0.1.0.0...
Preprocessing executable 'task-manager' for task-manager-0.1.0.0...
Building executable 'task-manager' for task-manager-0.1.0.0...
[1 of 1] Compiling Main             ( app/Main.hs,
↪  dist-newstyle/build/x86_64-linux/ghc-9.14.1/task-manager-0.1.0.0/x/task-m⌋
↪  anager/build/task-manager/task-manager-tmp/Main.o ) [Flags
↪  changed]
[2 of 2] Linking dist-newstyle/build/x86_64-linux/ghc-9.14.1/task-manager-0.1⌋
↪  .0.0/x/task-manager/build/task-manager/task-manager [Objects
↪  changed]
```

The first build configures the project: Cabal's solver checks all dependencies, verifies version constraints, and creates a build plan. Subsequent builds are incremental: only changed modules are recompiled.

Edit app/Main.hs to add a basic list command:

```haskell
{-# LANGUAGE OverloadedStrings #-}

module Main (main) where

import Data.Text (pack)
import qualified Data.Text.IO as TIO
import System.Environment (getArgs)

data Task = Task
  { taskName :: String
  , taskDone :: Bool
  } deriving (Show, Eq)

main :: IO ()
main = do
    args <- getArgs
    case args of
        [] -> printUsage
        ["list"] -> do
            tasks <- loadTasks
            TIO.putStrLn $ pack $ unlines $ map formatTask tasks
        ["add", name] -> do
            tasks <- loadTasks
            saveTasks $ tasks ++ [Task name False]
            TIO.putStrLn $ pack $ "Added: " ++ name
        _ -> printUsage

```

```haskell
printUsage :: IO ()
printUsage = do
    TIO.putStrLn "Task Manager v0.1.0"
    TIO.putStrLn "Usage: task-manager [list|add NAME]"

loadTasks :: IO [Task]
loadTasks = return []

saveTasks :: [Task] -> IO ()
saveTasks _ = return ()

formatTask :: Task -> String
formatTask t = (if taskDone t then "[x] " else "[ ] ")
               ++ taskName t
```

Run:

```
$ cabal run task-manager -- list
(one blank line: the task list is empty)

$ cabal run task-manager -- add "Buy groceries"
Added: Buy groceries
```

Notice how only Main.hs is recompiled on the second build because it was the only changed file. Cabal's incremental build tracking keeps development fast.

This establishes the basic development loop: edit code, cabal build (or cabal run, which builds implicitly), observe output, repeat. Throughout this book we will layer more sophistication on this workflow: testing, property checking, profiling, continuous integration, and effect-system architecture. But the core loop remains the same.

## Summary

You now have a working Haskell project built with modern tooling. We have installed GHC, Cabal, and HLS via GHCup. We have created a minimal executable project with proper Cabal configuration, dependency declarations, and warning flags. We have established the basic development loop.

The next chapter dives into how GHC actually compiles your code: the compilation pipeline, intermediate representations, optimization phases, and

the artifacts GHC produces. Understanding this pipeline is essential for debugging compilation issues, understanding optimization behavior, and making informed decisions about compiler flags.

# Chapter 2: How GHC Compiles Your Code

## The Compilation Pipeline

When you run cabal build on your project, Cabal invokes GHC to compile each Haskell module. GHC's compilation pipeline transforms your source code through several distinct intermediate representations before producing machine code. Understanding this pipeline is not academic curiosity: it explains why certain GHC flags matter, how optimization works, where type checking happens, and how to diagnose compilation problems.

The pipeline has five major phases:

1. **Frontend**: Lexing, parsing, renaming, and type checking produce GHC's Core language
2. **Core to Core**: Repeated simplification passes optimize the Core program
3. **Core to STG**: Translation to the Spineless Tagless G-machine intermediate language
4. **STG to Cmm**: Translation to C−, a low-level stack-based intermediate language
5. **Code Generation**: Compilation to native assembly or LLVM bitcode

Each phase operates on a well-defined intermediate representation. GHC can dump the output of each phase for inspection using flags like -ddump-parsed, -ddump-rn, -ddump-types, -ddump-simpl, -ddump-stg, -ddump-cmm, and -ddump-asm. These dumps are invaluable for understanding what GHC is doing to your code.

Let us walk through the pipeline with a concrete example. Create a simple module:

```haskell
module Example where

square :: Int -> Int
square x = x * x

sumSquares :: [Int] -> Int
sumSquares xs = sum (map square xs)
```

Compile and inspect the parsed AST:

```
$ ghc -ddump-parsed Example.hs
[1 of 1] Compiling Example          ( Example.hs, Example.o )

==================== Parser ====================
module Example where
square :: Int -> Int
square x = x * x
sumSquares :: [Int] -> Int
sumSquares xs = sum (map square xs)
```

The Parser section re-renders your source from the Haskell abstract syntax tree (HsSyn). This is a faithful representation of your program's structure, but not yet type-checked.

Next, rename and type-check. GHC skips recompiling (and therefore skips dumping) a module that is already up to date, so we add -fforce-recomp to force a fresh compilation:

```
$ ghc -fforce-recomp -ddump-types Example.hs
[1 of 1] Compiling Example          ( Example.hs, Example.o )
TYPE SIGNATURES
  square :: Int -> Int
  sumSquares :: [Int] -> Int
Dependent modules: []
Dependent packages: [(normal, base-4.22.0.0)]
```

After renaming (resolving variable references to their declarations) and type checking (inferring and verifying types), GHC produces a fully typed Core program. -ddump-types shows the settled type signatures and the packages the module depends on. To see the typed Core itself – with the I# constructor and GHC.Prim operations that form the underlying representation of Int [2] – dump it with -ddump-simpl and no optimization flag, as in the next section.

## Core: GHC's Intermediate Language

Core is GHC's primary intermediate language and the most important representation to understand. Core is a subset of Haskell that is strongly typed, lambda-calculus based, and designed for optimization. All your Haskell code, regardless of language extensions used, eventually becomes Core before further transformation.

Key properties of Core:

- **Strongly typed**: Every expression has a known, checked type
- **Lambda calculus foundation**: All control flow reduces to lambda abstractions and applications
- **Let-bound only**: No nested lambda bodies; everything is bound at top level via let
- **Algebraic data types only**: All types are built from constructors
- **Lazy**: By default, evaluation is lazy (thunks are created for deferred computation)

Let us see what Core looks like. Compile with -O and dump simplified Core:

```
$ ghc -O -ddump-simpl Example.hs 2>&1 | head -50
```

You will see something like:

```
square :: Int -> Int
[GblId, Arity=1, Str=<S!P(L)>, Cpr=1,
 Unf=Unf{Src=StableSystem, TopLvl=True,
         Value=True, ConLike=True, WorkFree=True, Expandable=True,
         Guidance=ALWAYS_IF(arity=1,unsat_ok=True,boring_ok=False), ...}]
square
  = \ (x_awC :: Int) -> GHC.Internal.Num.$fNumInt_$c* x_awC x_awC

$wgo1_r14W :: [Int] -> GHC.Internal.Prim.Int# -> Int
[GblId[StrictWorker([!])], Arity=2, Str=<1L><L>, Unf=OtherCon []]
$wgo1_r14W
  = \ (ds_s14D :: [Int]) (ww_s14G :: GHC.Internal.Prim.Int#) ->
      case ds_s14D of {
        [] -> GHC.Internal.Types.I# ww_s14G;
        : y_a14r ys_a14s ->
          case y_a14r of { GHC.Internal.Types.I# x_a14d ->
```

```
17            $wgo1_r14W
18              ys_a14s
19              (GHC.Internal.Prim.+# ww_s14G (GHC.Internal.Prim.*# x_a14d x_a14d))
20            }
21        }
22
23  sumSquares :: [Int] -> Int
24  [GblId, Arity=1, Str=<1L>, Cpr=1, Unf=Unf{...}]
25  sumSquares = \ (xs_azi :: [Int]) -> $wgo1_r14W xs_azi 0#
```

The metadata on each definition is crucial information:

- Arity indicates how many arguments the function takes before doing real work
- Str=<S!P(L)> indicates the strictness: the function is strict in its argument (S means the argument must be evaluated before being used)
- Unfolding information tells GHC when it should inline this function at call sites
- WorkFree=True and ConLike=True indicate the function is cheap and behaves like a constructor

The body shows Core's structure. Case expressions handle pattern matching. The I# constructor wraps the unboxed Int# primitive type. All operations are explicit; there is no syntactic sugar. Note how the simplifier has fused map square and sum into a single strict worker function ($wgo1_r14W) that accumulates the running total with unboxed Int# arithmetic (+# and *#).

Core is where GHC does its most important optimizations: inlining, strictness analysis, worker-wrapper transformation, specialization, and fusion. All of these operate on Core. The Core-to-Core simplifier runs many passes, each applying different optimizations, until a fixed point is reached.

Reading Core is an essential debugging and optimization skill. When your code runs slower than expected, or uses too much memory, or when GHC refuses to optimize in a way you anticipated, dumping Core with -ddump-simpl and reading the output tells you what GHC actually compiled, not what you wrote.

## The STG Machine and Register Allocation

After Core optimization, GHC translates to the Spineless Tagless G-machine (STG) language. STG is named after a theoretical machine model designed

for evaluating lazy functional programs efficiently. Unlike Core, which is mathematically oriented, STG is implementation-oriented: it models the actual runtime behavior of your program, including heap allocation, thunk evaluation, and lazy evaluation.

Key STG characteristics:

- **Explicit heap allocation**: Every data structure allocation is explicit
- **Thunks**: Deferred computations are represented as thunks that evaluate on first access
- **Register-based**: Uses a small set of virtual registers
- **Call conventions**: Explicit entry and exit points for functions

Compile and dump STG:

```
$ ghc -O -fforce-recomp -ddump-stg-from-core Example.hs 2>&1 | head -60
```

Sample output:

```
Example.square :: GHC.Internal.Types.Int -> GHC.Internal.Types.Int
[GblId, Arity=1, Str=<S!P(L)>, Cpr=1, Unf=OtherCon []] =
    \r [x_s14X] GHC.Internal.Num.$fNumInt_$c* x_s14X x_s14X;

...

$wgo1_r14W
  :: [GHC.Internal.Types.Int]
     -> GHC.Internal.Prim.Int# -> GHC.Internal.Types.Int
[GblId[StrictWorker([!])], Arity=2, Str=<1L><L>, Unf=OtherCon []] =
    \r [ds_s14Y ww_s14Z]
        case ds_s14Y of {
          [] -> GHC.Internal.Types.I# [ww_s14Z];
          : y_s151 [Occ=Once1!] ys_s152 [Occ=Once1] ->
              case y_s151 of {
              GHC.Internal.Types.I# x_s154 ->
              case *# [x_s154 x_s154] of $wgo1_sat_s155 [Occ=Once1] {
              __DEFAULT ->
              case +# [ww_s14Z $wgo1_sat_s155] of $wgo1_sat_s156 [Occ=Once1] {
              __DEFAULT -> $wgo1_r14W ys_s152 $wgo1_sat_s156;
              };
              };
              };
        };
```

STG is closer to the runtime than Core. The bracketed register names after \r (for example [x_s14X], or [ds_s14Y ww_s14Z] for multiple arguments) show how arguments are allocated to registers. The thunk structure and case analysis model how evaluation actually happens at runtime.

The STG compiler performs its own optimizations, including register allocation, thunk blackholing (preventing multiple evaluations of the same thunk), and unboxing of strict values.

## Code Generation and Object Files

After STG, GHC translates to C−, a very low-level stack-based language designed for the GHC runtime. C− has a small set of instructions for stack manipulation, heap allocation, function calls, and primitive operations. It is deliberately simple: just complex enough to express what the GHC runtime needs, simple enough to generate efficient code from.

From C−, GHC has two primary code generation backends:

1. **Native code generator** (-fasm, default): Generates assembly directly for the target architecture
2. **LLVM backend** (-fllvm): Generates LLVM bitcode, which LLVM optimizes and assembles

The LLVM backend is sometimes preferred for certain platforms or optimization profiles. It requires LLVM installed and must be compiled into GHC.

Dump the generated assembly:

```
1  $ ghc -O -fforce-recomp -ddump-asm Example.hs 2>&1 | head -40
```

You will see platform-specific assembly instructions. On x86-64 Linux, it will be x86-64 assembly with calls into the GHC runtime for heap operations, exception handling, and garbage collection integration.

The final output of compilation is one or more object files (.o) and an interface file (.hi). The object file contains compiled machine code linked against the GHC runtime. The interface file contains type information, export lists, and optimization metadata that other modules can use when they import this module.

## The Compilation Graph and Dependencies

In a real project with multiple modules, GHC compiles modules in dependency order. The compilation graph is determined by module imports: if module A imports module B, then B must be compiled (and its interface file available) before A can be compiled.

GHC tracks this dependency graph carefully. When you modify a module, GHC recompiles only that module and any modules that depend on it. This is how incremental compilation works.

Cabal manages this at a higher level. When you run cabal build, Cabal:

1. Determines which components need building
2. For each component, determines which modules need recompilation
3. Invokes GHC with the appropriate flags and import paths for each module
4. Links object files into executables or libraries

The dependency tracking is not perfect. GHC uses "fast interface checking" by default, which quickly determines if an interface file has changed. However, some changes (especially to optimization metadata like unfoldings or strictness information) may trigger more recompilation than strictly necessary. This is a trade-off between correctness and speed.

For multi-package projects, the dependency graph spans packages as well as modules. Cabal must build packages in topological order, ensuring each package's dependencies are available.

## GHC Artifact Files: Hi, O, and Beyond

GHC produces several types of artifact files during compilation. Understanding these files and their purposes helps with debugging, with understanding how GHC caches information, and with advanced build configurations.

**Interface files (.hi)**: Generated for every compiled module. An interface file contains:

- Type signatures of exported entities
- Unfolding information (for inlining in other modules)

- Strictness and laziness information
- Dependency information
- Pragmas and annotations

Interface files are critical for cross-module optimization. When module A inlines a function from module B, A reads B's .hi file to get the function's unfolding. Interface files are cached in dist-newstyle for Cabal builds.

**Object files (.o)**: Generated for every compiled module. These contain:

- Compiled machine code
- Symbol tables
- References to the GHC runtime system

Object files are later linked into executables or shared libraries.

**Dynamic objects (.dyn_o)**: When compiling with -dynamic-too or -fhc-core-dump, GHC may generate separate dynamic objects for use in GHCi or with dynamic linking.

**Compiled Haskell files (.ghci)**: GHCi can use precompiled interface files for faster loading.

Examine your project's artifacts:

```
$ find dist-newstyle -name "*.hi" -o -name "*.o" | head -20
dist-newstyle/build/x86_64-linux/ghc-9.14.1/task-manager-0.1.0.0/x/task-manag⌋
↪  er/build/task-manager/task-manager-tmp/Main.o
dist-newstyle/build/x86_64-linux/ghc-9.14.1/task-manager-0.1.0.0/x/task-manag⌋
↪  er/build/task-manager/task-manager-tmp/Main.hi
```

The path structure encodes: platform, GHC version, package name and version, component type (x for executable), and module name.

**Interface file contents**: You can inspect an interface file directly:

```
$ ghc --show-iface dist-newstyle/build/x86_64-linux/ghc-9.14.1/task-manager-0⌋
↪  .1.0.0/x/task-manager/build/task-manager/task-manager-tmp/Main.hi
```

This shows the full interface: exported symbols, their types, strictness information, and dependency declarations.

**Build artifacts management**: For clean builds:

```
$ cabal clean        # Remove all build artifacts
$ rm -rf dist-newstyle  # Manual removal
```

For partial cleanup (removing artifacts from an old GHC version, keeping the rest of the build cache):

```
$ rm -rf dist-newstyle/build/*/ghc-9.12.2
```

## Summary

GHC's compilation pipeline transforms your Haskell source code through five major phases: frontend processing to Core, Core-to-Core optimization, Core-to-STG translation, STG-to-C–, and final code generation. Each phase operates on a well-defined intermediate representation, and each can be inspected using dump flags.

Core is the most important intermediate language: it is where type-checked, semantically complete programs live and where most optimizations happen. Reading Core with -ddump-simpl is an essential skill for understanding what GHC has actually compiled.

The STG and C– stages bridge the gap between abstract Core and concrete machine code, modeling runtime behavior including lazy evaluation and heap allocation.

Interface files (.hi) are critical for incremental compilation and cross-module optimization. They cache type information, unfoldings, and strictness metadata.

In the next chapter we turn to GHC's type system: the kind system, type classes, language extensions, and how to use them effectively in production code.

# Chapter 3: GHC's Type System in Practice

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Types, Kinds, and the Kind System

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Type Classes: Structure and Resolution

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Essential Language Extensions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Advanced Extensions: GADTs, DataKinds, TypeFamilies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Type Error Diagnostics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Type System Pitfalls and Anti-patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

# Chapter 4: GHC Optimization and Compiler Flags

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Optimization Levels: -O, -O1, -O2

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Key Optimization Flags

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Profiling Flags and Their Interactions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Development vs Production Build Configurations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Inspecting Optimized Code with -ddump-simpl

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Optimization Gotchas: Float Invariants, Specialization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

# Chapter 5: The GHC Runtime System

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## The RTS Architecture

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Garbage Collection: Generational and Concurrent

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Lightweight Threads and the Scheduler

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Mutable State: MVars, TVars, STM

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## RTS Flags and Configuration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Memory Layout and Performance Implications

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

# Chapter 6: GHCi and Interactive Development

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## GHCi Basics and Essential Commands

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Loading and Reloading Modules

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Interactive Debugging with Breakpoints and Stepping

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Script Mode and :! Commands

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## GHCi Configuration and Startup Scripts

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Interactive Development Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

# Chapter 7: GHC Diagnostics and Warnings

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Warning Flags and Categories

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Essential Warnings for Production Code

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Configuring Warnings in Cabal and GHC Files

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Reading GHC Type Errors Systematically

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Common Error Patterns and Fixes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Setting Up Your Project for Maximum Diagnostics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

# Chapter 8: Cabal Package Management Fundamentals

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## What a Cabal Package Is

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Package Description Format

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Dependencies and Version Constraints

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## The Solver and Dependency Resolution

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Package Index and Hackage

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Cabal Config and Global Settings

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

# Chapter 9: Building Projects with Cabal

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Components: Libraries, Executables, Tests, Benchmarks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Build Dependencies and Configuration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Cabal Flags and Feature Management

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Build Scripts and Custom Setup

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Incremental Builds and Caching

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Build Output and Generated Files

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

# Chapter 10: Advanced Cabal: Large Projects and Multi-Package Sets

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Multi-Package Projects and Cabal Workspaces

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Local Package Dependencies and Path Constraints

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Solver Constraints and Pins

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Managing Dependencies Across Many Packages

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Splitting Large Projects: Library vs Application Boundaries

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Real-World Multi-Package Project Structures

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

# Chapter 11: Reproducible Builds, CI/CD, and Project Hygiene

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Reproducible Builds and Freezes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Haskell Stack vs Cabal: When Each Makes Sense

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Continuous Integration for Haskell Projects

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Cross-Compilation and Platform-Specific Builds

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Documentation Generation with Haddock

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Keeping Dependencies Updated Safely

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

# Chapter 12: The Haskell Language Server

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## What HLS Does and Why It Matters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Installing and Configuring HLS

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Diagnostics and Real-Time Error Checking

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Code Completion and Navigation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Code Actions and Refactoring

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Expression Evaluation and Hover Information

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

# Chapter 13: Editor Integration and HLS Workflows

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## VS Code and haskell-vscode

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Neovim and HLS Integration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Emacs and HLS Configuration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Shared HLS Settings Across Editors

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## HLS Performance Tuning for Large Projects

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Troubleshooting Common Editor Integration Issues

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

# Chapter 14: HLS Architecture and Internals

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## The GHC API and HLS's Use of It

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## How HLS Parses and Type-Checks Your Code

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Caching and Incremental Processing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## The HLS Plugin System

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Performance Characteristics and Bottlenecks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Extending HLS with Custom Plugins

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

# Chapter 15: Purity, Effects, and the IO Monad

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Purity and the No-Side-Effects Guarantee

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## IO as a Computations Type, Not a Tag

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## The Monad Abstraction

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Do Notation and Effect Sequencing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Practical IO Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

# Chapter 16: Type Classes and Effect Abstraction

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Effect Abstraction via Type Classes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## The Hierarchy: Functor, Applicative, Monad

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## MonadTrans and the Lifting Problem

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Common Effect Type Classes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Designing Your Own Effect Classes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## When Effect Abstraction Helps and Hurts

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

# Chapter 17: Monad Transformers and MTL-Style Programming

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Monad Transformers: The Idea

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Common Transformers: ReaderT, StateT, ExceptT, WriterT

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Building Effect Stacks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## The MTL Pattern and Type Signatures

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## ReaderT-Based Architectures for Services

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## MTL Complexity and Maintenance Costs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

# Chapter 18: Capability-Based Effects and Modern Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## What Are Capabilities?

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Designing Capability-Based APIs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## The io-universe-family and Similar Approaches

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Testing with Capability-Based Designs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Comparing Capabilities vs MTL

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Real-World Capability-Based Systems

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

# Chapter 19: Algebraic Effects and Freer Approaches

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Algebraic Effects: The Concept

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Free Monads and Freer Monads

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## The extensible-effects Library

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Polysemy: Design and Trade-offs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## The effectful Library and Effect Handlers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Choosing an Effects Library

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

# Chapter 20: Comparing Effect Systems and Choosing an Approach

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Evaluation Criteria for Effect Systems

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Direct IO vs Abstracted Effects

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## MTL vs Capabilities vs Algebraic Effects: Head to Head

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Performance Comparison and Benchmarks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Debugging and Observability Trade-offs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Recommendations by Project Type and Size

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

# Chapter 21: Testing, Property Testing, and Quality Assurance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Testing Philosophy in Haskell

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Unit Testing with Hspec

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Tasty as a Test Harness

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Property-Based Testing with QuickCheck

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Hedgehog: The Modern Alternative

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Integration Testing with Real Dependencies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

# Chapter 22: Profiling, Benchmarking, and Performance Tuning

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## CPU Profiling with -prof and -auto-all

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Reading GHC Profile Output

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Memory and Heap Profiling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Runtime Statistics and RTS Reporting

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Benchmarking with Criterion

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Performance Tuning Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

# Chapter 23: Debugging, Observability, and Production Practices

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Debugging Strategies in a Pure Language

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Logging Architectures for Haskell Services

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Tracing and OpenTelemetry Integration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Handling Errors and Failures Gracefully

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Production Configuration and Health Checks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Incident Response for Haskell Services

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

# Chapter 24: Troubleshooting Case Studies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Dependency Resolution Failures

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Confusing Type Errors: A Methodical Approach

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Compilation Errors from Missing Extensions and Flags

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## HLS Not Working: Common Causes and Fixes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Slow Compilation: Diagnosis and Remedies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Runtime Crashes and Memory Issues

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

# Chapter 25: Conclusion: The Future of Haskell Tooling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## What Has Changed in the Last Decade

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Ongoing Work in GHC

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## The Cabal and Build System Future

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## HLS and Developer Experience

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Effect Systems: Where the Ecosystem Is Heading

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

## Haskell in the Professional Landscape

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.

# References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernhaskelltooling.