STUNIR Haskell Pipeline Status Report

Date: January 31, 2026
Version: Week 9 Assessment
Status: ⚠️ NOT FUNCTIONAL - Toolchain Not Available

Executive Summary

The Haskell pipeline implementation exists in the codebase with comprehensive structure, but **cannot be tested or used** due to missing Haskell toolchain on the development system.

Key Findings:
- ✅ Haskell source code present at `tools/haskell/`
- ✅ Build configuration exists (`stunir-tools.cabal`)
- ❌ No Haskell compiler (GHC) installed
- ❌ No Cabal build tool installed
- ❌ No Stack build tool installed
- ❓ Implementation appears to be stub/skeleton code
- ❓ Unknown if implementation is complete

Pipeline Components

1. Spec to IR (`SpecToIR.hs`)

Location: `tools/haskell/src/SpecToIR.hs`

Status: ⚠️ Skeleton Implementation

Key Observations:

```haskell
generateIR :: Value -> IRModule
generateIR spec = IRModule
  { moduleName = "unnamed_module"
  , moduleVersion = "1.0.0"
  , moduleFunctions = []
  }
```

- ❌ Hardcoded module name
- ❌ Always returns empty function list
- ❌ Does not parse spec JSON structure
- ✅ SHA-256 hashing infrastructure present
- ✅ Command-line interface implemented
- ✅ Output format matches `stunir_ir_v1` schema

Features:
- Version flag support: `--version`

- Output redirection: `-o OUTPUT`
- Error handling framework
- JSON parsing infrastructure (using `aeson` library)

2. IR to Code (`IRToCode.hs`)

Location: `tools/haskell/src/IRToCode.hs`

Status: ⚠️ Skeleton Implementation

Key Observations:
- ✅ Command-line interface complete
- ✅ IR parsing framework exists
- ✅ Target language selection (`-t TARGET`)
- ❓ Code emission logic in `STUNIR.Emitter` module (not inspected)
- ✅ Error handling present

Features:
- Version flag: `--version`
- Target selection: `-t c` , `-t python` , etc.
- Output redirection: `-o OUTPUT`
- IR manifest parsing

Build Configuration

Cabal Files

Primary: `tools/haskell/stunir-tools.cabal`

Executables:
1. `stunir_spec_to_ir` - Spec to IR converter
2. `stunir_ir_to_code` - IR to code emitter

Dependencies:

```
base ^>=4.16           # Haskell base (GHC 9.2+)
aeson >=2.0            # JSON parsing/encoding
bytestring >=0.11      # Binary data handling
text >=1.2             # Text processing
cryptonite >=0.30      # Cryptographic hashing (SHA-256)
memory >=0.18          # Memory operations
containers >=0.6       # Data structures
mtl >=2.2              # Monad transformers (ir_to_code only)
```

Test Suite:

```
test-suite stunir-emitters-test
  type: exitcode-stdio-1.0
  main-is: Main.hs
  build-depends:
    - hspec >=2.7       # BDD testing framework
    - QuickCheck >=2.14 # Property-based testing
```

GHC Compiler Flags:
- `-Wall` : All warnings enabled
- `-O2` : Maximum optimization
- `-threaded` : Multi-threading support

Toolchain Requirements

Minimum Requirements

GHC (Glasgow Haskell Compiler):
- Version: 9.2 or higher (for `base ^>=4.16`)
- Installation size: ~1.2 GB
- Platform: Linux x86_64 / ARM64

Cabal (Build Tool):
- Version: 3.0 or higher
- Bundled with GHC or separate install

Alternative: Stack:
- Version: 2.9+ recommended
- Manages GHC versions automatically
- Requires `stack.yaml` configuration file (currently missing)

Installation Options

Option 1: GHCup (Recommended)

```
# Install GHCup (Haskell version manager)
curl --proto '=https' --tlsv1.2 -sSf https://get-ghcup.haskell.org | sh

# Install GHC 9.2.8 and Cabal
ghcup install ghc 9.2.8
ghcup install cabal 3.8.1.0
ghcup set ghc 9.2.8
```

Option 2: System Package Manager

```
# Ubuntu/Debian
sudo apt-get update
sudo apt-get install ghc cabal-install

# Note: May install older versions
```

Option 3: Stack

```
# Install Stack
curl -sSL https://get.haskellstack.org/ | sh

# Build with Stack (creates stack.yaml needed)
cd tools/haskell
stack init
stack build
```

Build Process (Once Toolchain Available)

```
cd /home/ubuntu/stunir_repo/tools/haskell

# Update package index
cabal update

# Build executables
cabal build all

# Run tests
cabal test

# Install to ~/.cabal/bin
cabal install

# Locate binaries
find . -name "stunir_spec_to_ir" -type f
find . -name "stunir_ir_to_code" -type f
```

Module Structure

Core Types (`STUNIR.Types`)

- `IRModule` - Top-level IR structure
- Type definitions for IR components

Hashing (`STUNIR.Hash`)

- `sha256JSON` - Deterministic SHA-256 hashing
- Uses `cryptonite` library

IR Handling (`STUNIR.IR`)

- `irToJSON` - Convert IR to JSON
- `parseIR` - Parse JSON to IR

Code Emission (`STUNIR.Emitter`)

- `emitCode` - Generate target language code
- Multi-target support

Emitter Modules (Extensive)

Structure: `STUNIR.SemanticIR.Emitters.*`

Categories:
1. **Core**:
- `Core.Embedded` - Embedded systems (ARM, AVR, MIPS)
- `Core.GPU` - GPU code generation
- `Core.WASM` - WebAssembly

- `Core.Assembly` - Low-level assembly
- `Core.Polyglot` - Multi-language targets

1. **Language Families**:
   - `LanguageFamilies.Lisp` - Lisp dialects
   - `LanguageFamilies.Prolog` - Logic programming

2. **Specialized**:
   - Business logic
   - FPGA synthesis
   - Grammar/Lexer/Parser generation
   - Expert systems
   - Constraint solving
   - Functional programming
   - OOP
   - Mobile platforms
   - Scientific computing
   - Bytecode generation
   - Systems programming
   - Planning/scheduling
   - AsmIR
   - BEAM (Erlang VM)
   - ASP (Answer Set Programming)

Total Emitter Modules: 17+ specialized emitters

Test Infrastructure

Test Files

Location: `tools/haskell/test/`

Test Suites:
1. `STUNIR.SemanticIR.Emitters.BaseSpec` - Base emitter tests
2. `STUNIR.SemanticIR.Emitters.CoreSpec` - Core emitter tests
3. `STUNIR.SemanticIR.Emitters.LanguageFamiliesSpec` - Lisp/Prolog tests
4. `STUNIR.SemanticIR.Emitters.SpecializedSpec` - Specialized emitter tests

Framework: Hspec + QuickCheck
- **Hspec**: BDD-style testing (describe/it blocks)
- **QuickCheck**: Property-based testing (randomized inputs)

Cannot Run: No Haskell toolchain installed

Comparison to Other Pipelines

Feature	Rust	SPARK	Python	Haskell
Toolchain Available	✅	✅	✅	❌
Builds Successfully	✅	✅	N/A	❓
Spec to IR Works	✅	✅	✅	❓
IR to Code Works	✅	✅	✅	❓
Multi-File Support	❌	❌	✅	❓
Function Bodies	❌	❌	❌	❓
Test Suite Runs	✅	❓	✅	❌
Code Quality	High	Highest	Reference	Unknown
Type Safety	Strong	Verified	Dynamic	Strong
Formal Verification	No	Yes	No	No

Implementation Completeness Assessment

Evidence of Incomplete Implementation

SpecToIR.hs:

```
generateIR :: Value -> IRModule
generateIR spec = IRModule
  { moduleName = "unnamed_module"
  , moduleVersion = "1.0.0"
  , moduleFunctions = []  -- ❌ STUB: Always empty
  }
```

This is clearly a placeholder implementation. Compare to Python's `spec_to_ir.py` which:
1. Parses `"module"` field from spec JSON
2. Extracts `"functions"` array

3. Converts each function to IR format
4. Processes arguments, return types, steps

Expected Full Implementation:

```
generateIR :: Value -> IRModule
generateIR spec =
  let moduleName' = extractModuleName spec
      version = extractVersion spec
      functions = extractFunctions spec  -- Parse functions array
      irFuncs = map convertFunctionToIR functions
  in IRModule
    { moduleName = moduleName'
    , moduleVersion = version
    , moduleFunctions = irFuncs
    }
```

Likelihood of Working Implementation

Probability: **<20%**

Reasons:
1. Core conversion logic is stubbed
2. No evidence of spec parsing beyond basic JSON decode
3. Function list always empty
4. Module name hardcoded
5. No indication of step generation
6. Emitter module contents not verified

What Would Need to Be Implemented

1. **Spec Parsing**:
   - Extract module metadata
   - Parse functions array
   - Parse function signatures (name, args, return type)
   - Parse function bodies (statements/steps)

2. **IR Generation**:
   - Convert function definitions to IR format
   - Generate step arrays with proper `op` types
   - Map data types (spec types → IR types)
   - Preserve call chains and dependencies

3. **Code Emission**:
   - Verify emitter modules are complete
   - Template rendering for each target language
   - Type mapping (IR types → target language types)
   - Step translation (IR ops → target code)

4. **Testing**:
   - Unit tests for spec parsing
   - IR generation validation
   - Code emission verification
   - End-to-end pipeline tests

Recommendations

Short-Term (Week 9)

❌ **DO NOT ATTEMPT**:
- Installing Haskell toolchain (1.2+ GB, complex setup)
- Building Haskell pipeline
- Testing Haskell implementation

Rationale:
- Other pipelines (Rust, SPARK, Python) are functional
- Week 9 priorities are more impactful:
- Function body emission (critical feature)
- Multi-file support (completes existing pipelines)
- SPARK step generation improvement
- Risk of toolchain installation issues
- Likely incomplete implementation

Medium-Term (Post-v1.0.0)

If Haskell Pipeline Is Prioritized:

1. **Install Toolchain** (1-2 hours)
   - Use GHCup for version management
   - Install GHC 9.2.8+
   - Install Cabal 3.8+
   - Verify with `ghc --version` and `cabal --version`

2. **Attempt Build** (30 minutes)
   ```
   bash
     cd tools/haskell
     cabal update
     cabal build all 2>&1 | tee build.log
   ```
   - Document any errors
   - Check dependency resolution

3. **Inspect Implementation** (1-2 hours)
   - Read all source files in `src/`
   - Verify emitter modules are complete
   - Compare logic to Python reference
   - Identify missing features

4. **Run Tests** (if build succeeds)
   ```
   bash
     cabal test
   ```
   - Document test results
   - Compare to Rust/SPARK/Python test coverage

5. **Complete Implementation** (Est. 2-4 weeks)
   - Implement full spec parsing
   - Complete IR generation logic
   - Verify code emission for all targets

   - Add multi-file support
   - Implement function body emission
   - Match Python reference behavior

6. **Integration** (1 week)
   - Update `scripts/build.sh` with Haskell detection
   - Add to CI/CD pipeline
   - Document in README
   - Add to pipeline comparison

Long-Term Considerations

Benefits of Completing Haskell Pipeline:
- ✅ Strong type safety (compile-time guarantees)
- ✅ Lazy evaluation (efficiency)
- ✅ Extensive emitter module structure (17+ targets)
- ✅ Property-based testing (QuickCheck)
- ✅ Academic credibility
- ✅ Functional programming paradigm diversity

Drawbacks:
- ❌ Large toolchain requirement (~1.2 GB)
- ❌ Additional maintenance burden
- ❌ Duplicate functionality (Rust/SPARK/Python already work)
- ❌ Limited community adoption for infrastructure tools
- ❌ Longer build times vs Rust/SPARK

Strategic Question:
Is a 4th implementation language justified, or should resources focus on:
- Completing existing pipelines to 100%?
- Adding new features (function bodies, multi-file)?
- Expanding test coverage?
- Improving documentation?
- Building example applications?

Week 9 Decision

Status for Week 9 Completion Report

Haskell Pipeline: ⚠️ **NOT FUNCTIONAL**

Reason: Toolchain not available, implementation incomplete

Impact on Project Goals:
- ❌ Cannot achieve “4/4 pipelines functional” goal
- ✅ Can achieve “3/4 pipelines functional + 1 documented”
- ✅ Other Week 9 priorities remain achievable

Recommendation:
Document status and defer Haskell pipeline work to post-v1.0.0

Focus Week 9 efforts on:
1. ✅ Function body emission (critical feature gap)
2. ✅ Multi-file support for Rust/SPARK (parity with Python)
3. ✅ SPARK step generation improvements
4. ✅ End-to-end testing of functional pipelines
5. ✅ Comprehensive documentation

Conclusion

The Haskell pipeline is **architecturally impressive** with extensive emitter modules and strong type safety, but is **not currently usable** due to:

1. Missing Haskell toolchain
2. Apparent stub implementation in core logic
3. Unverified emitter module completeness

For v1.0.0 release:
- Focus on **completing the 3 functional pipelines** (Rust, SPARK, Python)
- Implement **critical missing features** (function bodies, multi-file)
- Document Haskell pipeline as **future work**

Post-v1.0.0:
- Evaluate strategic value of 4th implementation language
- If valuable, allocate 3-5 weeks for:
- Toolchain setup
- Implementation completion
- Testing and integration
- Documentation

Status Summary: 📋 **DOCUMENTED** | 🚫 **NOT TESTED** | ⏸️ **DEFERRED**