

Liquid PureScript Feasibility Study

Status: active **Category:** research **Created:** 2026-01-29 **Author:** Claude (research session)

Executive Summary

Building a Liquid Haskell-style refinement type system for PureScript is **technically feasible** but represents a significant engineering effort. The key enablers are:

- 1. **CoreFn intermediate representation** - PureScript already outputs a well-documented IR
- 2. **purescript-z3 bindings** - Z3 SMT solver is already accessible from PureScript
- 3. **Backend-agnostic architecture** - Multiple backends (JS, Erlang, etc.) consume CoreFn

The main challenges are:

- 1. No compiler plugin system (would need to fork compiler or build external tool)
- 2. Type erasure complications (which refinements survive to runtime?)
- 3. FFI boundary handling (foreign code can't be verified)

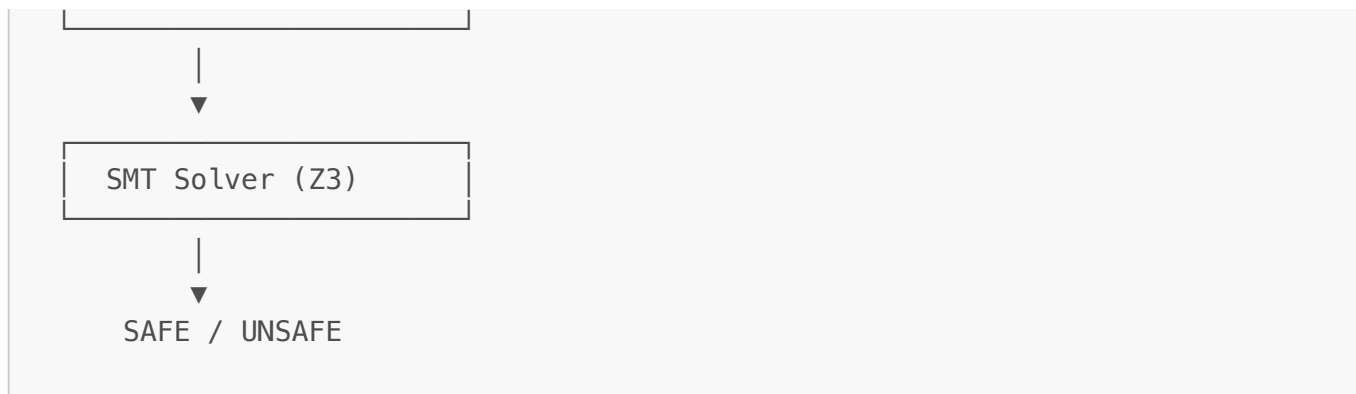
Recommended approach: Build as an external tool consuming CoreFn, not as a compiler modification.

Part 1: How Liquid Haskell Works

Architecture Overview

Liquid Haskell operates as a GHC plugin, hooking into the `typecheckResultAction` phase:





Key Technical Choices

1. **Unoptimized Core:** LH requires `-O0` because optimizations can change type representations
2. **Horn Clauses:** Verification conditions are encoded as Horn clauses for SMT
3. **Annotation Syntax:** Refinements live in specially-formatted Haskell comments
4. **Decidable Logic:** Uses QF-EUFLIA (quantifier-free equality, uninterpreted functions, linear integer arithmetic)

What LH Verifies

```

{-@ type Pos = { v:Int | v > 0 } @-}
{-@ abs :: Int -> Pos @-}
abs :: Int -> Int
abs n | n > 0      = n
      | otherwise = -n
  
```

LH generates constraints:

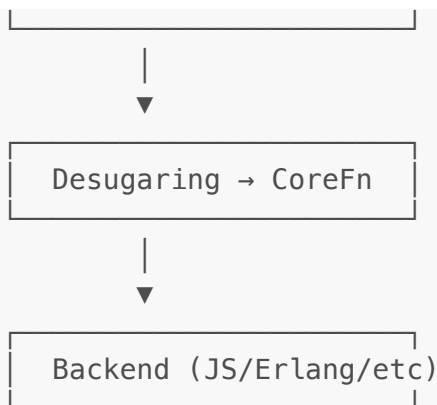
- If `n > 0`, prove `n > 0` (trivial)
- If `not (n > 0)`, prove `-n > 0` (follows from `n <= 0 ∧ n ≠ 0` for non-zero inputs)

Part 2: PureScript's Compiler Architecture

CoreFn: The Key Enabler

PureScript compiles to CoreFn, a simplified functional IR:





CoreFn is:

- Well-documented and stable
- Dumped as JSON (`purs compile --codegen corefn`)
- Used by multiple backends (purerl, purescript-backend-optimizer)
- Simpler than source (no type classes, no do-notation sugar)

No Plugin System

Unlike GHC, PureScript doesn't have a compiler plugin system. Options:

1. **Fork the compiler** - Invasive, maintenance burden
2. **External tool consuming CoreFn** - Clean, independent development
3. **Build into Spago** - Would still need to consume CoreFn

Recommendation: External tool is the pragmatic path.

Type Information in Externs

`externs.json` files contain exported type signatures. Combined with CoreFn, this provides enough information for refinement checking.

Part 3: Existing PureScript Building Blocks

purescript-z3

[purescript-z3](#) provides FFI bindings to Z3:

```

import Z3 as Z3

solve :: Effect (Maybe Solution)
solve = Z3.run do
  x <- Z3.int "x"
  y <- Z3.int "y"
  Z3.assert (x `Z3.gt` Z3.intVal 0)
  Z3.assert (y `Z3.eq` (x `Z3.mul` Z3.intVal 2))
  Z3.withModel \model -> do
    xVal <- Z3.eval model x
  
```

```
yVal <- Z3.eval model y
pure { x: xVal, y: yVal }
```

Status: Partial bindings, but sufficient for constraint solving. Uses Z3's WASM build.

purescript-refined

[purescript-refined](#) provides runtime-checked refinement types:

```
type DiceRoll = Refined (FromTo D1 D6) Int

validRoll :: Either RefinedError DiceRoll
validRoll = refine 5 -- Right DiceRoll

invalidRoll :: Either RefinedError DiceRoll
invalidRoll = refine 8 -- Left (FromToError 1 6 8)
```

Limitation: Runtime checking only. A Liquid system would verify these statically.

purescript-corefn

[purescript-corefn](#) provides PureScript types for working with CoreFn JSON:

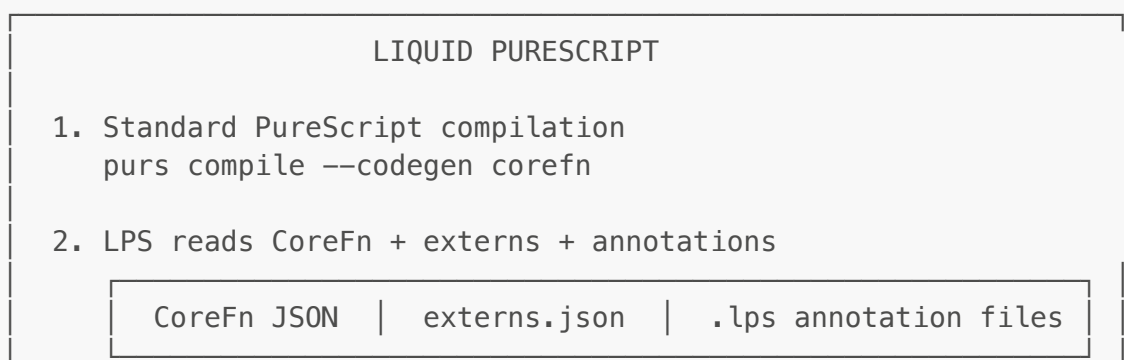
```
import PureScript.CoreFn (Module, readModuleJSON)

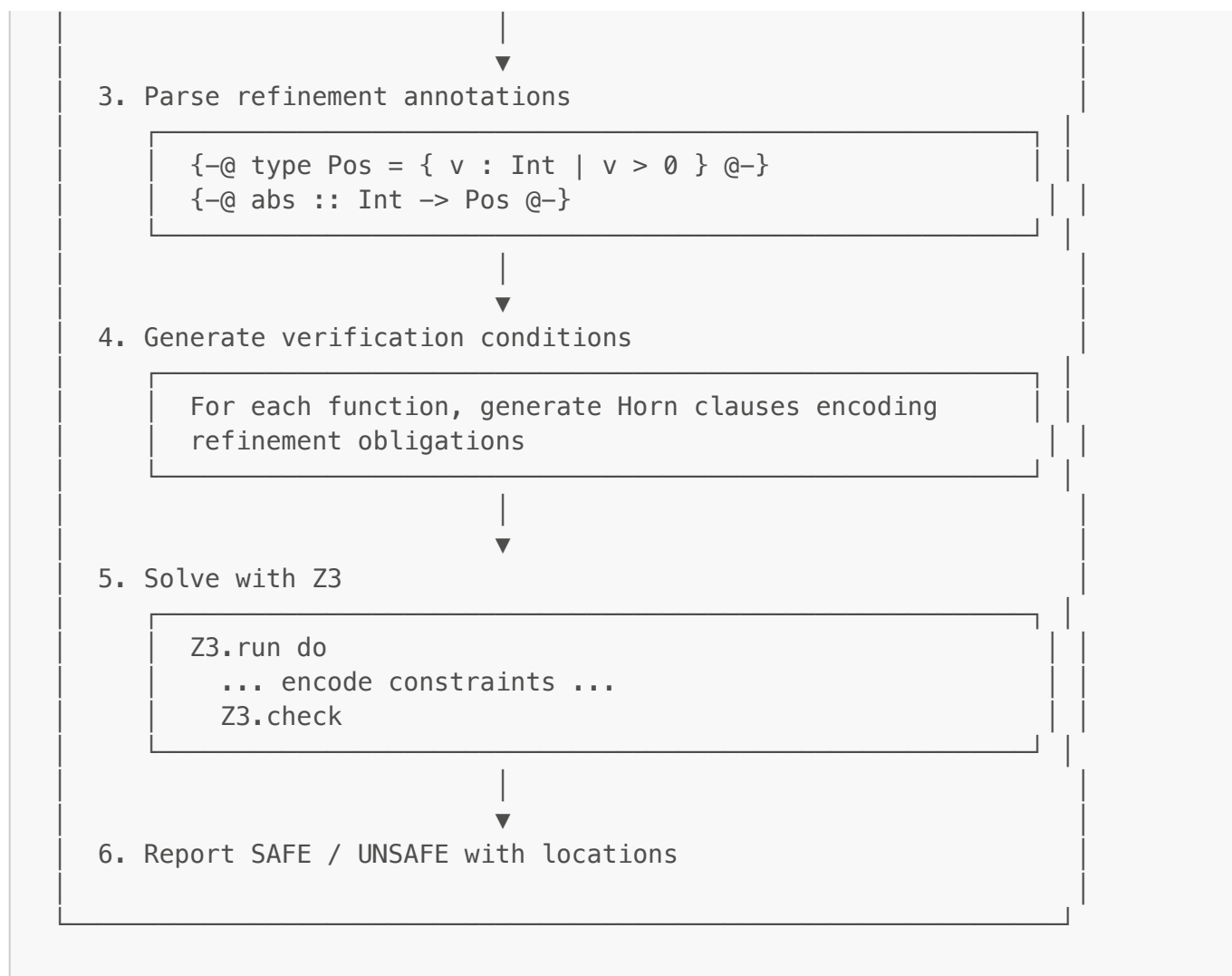
loadModule :: String -> Effect Module
loadModule path = do
  json <- readTextFile path
  pure (readModuleJSON json)
```

Status: Useful for parsing CoreFn, though may need updates for recent compiler versions.

Part 4: Architecture for Liquid PureScript

Proposed Pipeline





Annotation Format

Following LH's lead, annotations in specially-marked comments:

```

-- .purs file
module MyModule where

{-@ type NonEmpty a = { v : Array a | length v > 0 } @-}

{-@ head :: NonEmpty a -> a @-}
head :: forall a. Array a -> a
head xs = unsafePartial (Array.head xs)

{-@ filter :: (a -> Boolean) -> Array a -> Array a @-}
filter :: forall a. (a -> Boolean) -> Array a -> Array a
filter = Array.filter

-- This would be UNSAFE - filter might return empty array
{-@ badHead :: Array Int -> Int @-}
badHead xs = head (filter (_ > 0) xs)  -- ERROR: Cannot prove non-empty

```

Alternative: Separate Spec Files

Like LH's `.spec` files:

```
-- MyModule.lps
module MyModule where

type NonEmpty a = { v : Array a | length v > 0 }
type Pos = { v : Int | v > 0 }

head :: NonEmpty a -> a
abs :: Int -> Pos
```

Advantage: No changes to `.purs` files, cleaner separation.

Part 5: Technical Challenges

Challenge 1: Type Erasure

PureScript erases types at runtime. For refinements:

```
{-@ type Pos = { v : Int | v > 0 } @-}

{-@ add :: Pos -> Pos -> Pos @-}
add :: Int -> Int -> Int
add x y = x + y
```

At runtime, `Pos` is just `Int`. The refinement exists only at verification time. This is fine for pure code but problematic for FFI boundaries.

Solution: Treat FFI as trust boundary. Foreign functions get assumed types.

Challenge 2: Higher-Order Functions

Refinements on function arguments:

```
{-@ map :: (a -> b) -> Array a -> Array b @-}
{-@ map :: (Pos -> Pos) -> NonEmpty Pos -> NonEmpty Pos @-} -- More
specific
```

LH handles this with "abstract refinements" - refinement variables that can be instantiated.

Complexity: Significant. This is where LH research papers get deep.

Challenge 3: Records and Row Types

PureScript's row-polymorphic records are more flexible than Haskell's:

```
type Person r = { name :: String, age :: Int | r }
```

How do refinements interact with row polymorphism?

```
{-@ type Adult r = { name :: String, age :: { v : Int | v >= 18 } | r }
@-}
```

Status: Novel research territory. No existing system handles this well.

Challenge 4: Type Classes

Refinements that vary by instance:

```
class Monoid a where
  mempty :: a
  mappend :: a -> a -> a

-- For Array: mempty = [] (length 0)
-- For NonEmptyArray: mempty = [default] (length 1)
```

LH Approach: Bounded refinement types, instance-specific specs.

Challenge 5: Laziness

Haskell is lazy; LH has complex handling for divergence. PureScript is strict, which simplifies things:

```
-- In strict PureScript, this always diverges:
loop :: forall a. a
loop = loop

-- In lazy Haskell, it only diverges if forced
```

Advantage: PureScript's strictness makes verification simpler.

Part 6: Implementation Roadmap

Phase 1: Minimal Viable Verifier (4-6 weeks)

Goal: Verify refinements on simple functions (no HOF, no polymorphism)

Deliverables:

1. CoreFn parser (or update purescript-corefn)
2. Annotation parser (comment format or separate files)

3. VC generator for basic expressions
4. Z3 integration for constraint solving
5. Error reporting with source locations

Demo: Verify `abs :: Int -> Pos` style functions

Phase 2: Polymorphism and Records (6-8 weeks)

Goal: Handle parametric polymorphism and records

Deliverables:

1. Abstract refinements for polymorphism
2. Record refinement syntax
3. Measure definitions (e.g., `length` for arrays)
4. Pre/post conditions on record fields

Demo: Verify `head :: NonEmpty a -> a`

Phase 3: Higher-Order Functions (8-12 weeks)

Goal: Verify functions taking function arguments

Deliverables:

1. Refinement inference for lambdas
2. Abstract refinement instantiation
3. Contravariant argument handling
4. Fix point inference for recursive functions

Demo: Verify `map`, `filter`, `fold` with refinements

Phase 4: Type Classes (Research)

Goal: Instance-specific refinements

Deliverables:

1. Bounded refinement types
2. Instance specs
3. Coherent refinement inheritance

Status: This requires genuine research, not just engineering.

Part 7: Comparison with Alternatives

Option A: Liquid PureScript (this document)

Approach: External tool, SMT-based verification **Pros:** Proven approach (LH), automatic verification, decidable **Cons:** Complex implementation, limited expressiveness

Option B: Full Dependent Types in Compiler

Approach: Modify PureScript compiler to support dependent types **Pros:** Full expressiveness, no separate tool **Cons:** Massive compiler change, breaks backward compatibility

From [GitHub issue #2214](#):

"implement a typechecker that supports dependent types along with records and typeclasses" - described as "the trivial task" (sarcastically)

Status: Closed as out-of-scope research project.

Option C: Idris Backend for JavaScript

Approach: Use Idris (dependently typed language) with JS backend **Pros:** Already exists, full dependent types **Cons:** Different language, immature JS backend

Option D: purescript-refined (Runtime Checking)

Approach: Library-based runtime checking **Pros:** Already works, no compiler changes **Cons:** Runtime cost, no static guarantees

Recommendation

Start with Option A (Liquid PureScript) because:

- 1. Proven approach from Liquid Haskell
- 2. External tool = independent development
- 3. Automatic verification = lower barrier for users
- 4. Can target important subset (non-empty arrays, positive numbers, etc.)

Part 8: Integration with Playground Vision

The typed feedback loop from the playground research becomes even more powerful with refinements:

CODE	REFINEMENTS
<pre>xs = [1, 2, 3]</pre>	<pre>xs : NonEmpty Int ✓ length = 3 > 0</pre>
<pre>ys = filter (_ > 5) xs</pre>	<pre>ys : Array Int △ might be empty</pre>
<pre>z = head ys</pre>	<pre>z : ERROR x head requires NonEmpty</pre>
<pre>-- Fix: add guard z = case ys of [] -> Nothing _ -> Just (head ys)</pre>	<pre>z : Maybe Int ✓ pattern handles empty</pre>

The playground doesn't just show types - it shows **what properties the types guarantee**.

Resources

Liquid Haskell

- [LiquidHaskell Homepage](#)
- [LH Tutorial](#)
- [LH as GHC Plugin \(Well-Typed\)](#)
- [LH ICFP'14 Paper \(PDF\)](#)

PureScript Internals

- [CoreFn source](#)
- [purescript-corefn library](#)
- [purescript-backend-optimizer](#)
- [Dependent Types Discussion](#)

SMT and Z3

- [purescript-z3](#)
- [Z3 JavaScript API](#)
- [SMT-LIB Standard](#)

Related Work

- [Refinement Types for TypeScript \(PDF\)](#)
 - [purescript-refined](#)
-

Conclusion

Liquid PureScript is feasible. The hardest part isn't "can we do it" but "where do we stop":

Scope	Effort	Value
Simple numeric refinements	4-6 weeks	Catch div-by-zero, bounds errors
Non-empty arrays	6-8 weeks	Eliminate partial function errors
Full polymorphism	8-12 weeks	General safety guarantees
Type classes	Research	Instance-specific invariants

The sweet spot is probably **Phase 2** - enough to catch real bugs (non-empty arrays, positive numbers, valid indices) without the research-level complexity of full dependent types.

Combined with the playground vision, this would give PureScript developers something no other JavaScript-targeting language has: **a tight feedback loop with static guarantees about data properties**.