

1. Introduction

1.1. Well-Typed Programs ...

1.2. Refinement Types

2. Refinement Types

2.1. Defining Types

2.2. Errors

2.3. Subtyping

2.4. Writing Specifications

2.5. Refining Function Typ...

2.6. Dependent Refinements

3. Refined Datatypes

3.1. Sparse Vectors

4. Case Study: Okasaki's Lazy

Queues

4.1. Queues

4.2. Sized Lists

4.3. Queue Type

4.4. Queue Operations

4.5. Recap of everything!

5. Cheat Sheet

5.1. Specifications

5.2. Alias

5.3. Liquid types in Datay...

5.1. Specifications

5.2. Alias

5.3. Liquid types in Datay...

5.1. Specifications

5.2. Alias

5.3. Liquid types in Datay...

5.1. Specifications

5.2. Alias

5.3. Liquid types in Datay...

5.1. Specifications

5.2. Alias

5.3. Liquid types in Datay...

5.1. Specifications

5.2. Alias

5.3. Liquid types in Datay...

5.1. Specifications

5.2. Alias

5.3. Liquid types in Datay...

5.1. Specifications

5.2. Alias

5.3. Liquid types in Datay...

5.1. Specifications

5.2. Alias

5.3. Liquid types in Datay...

5.1. Specifications

5.2. Alias

5.3. Liquid types in Datay...

5.1. Specifications

5.2. Alias

5.3. Liquid types in Datay...

5.1. Specifications

5.2. Alias

5.3. Liquid types in Datay...

5.1. Specifications

5.2. Alias

5.3. Liquid types in Datay...

5.1. Specifications

5.2. Alias

5.3. Liquid types in Datay...

5.1. Specifications

5.2. Alias

5.3. Liquid types in Datay...

5.1. Specifications

5.2. Alias

5.3. Liquid types in Datay...

5.1. Specifications

5.2. Alias

5.3. Liquid types in Datay...

5.1. Specifications

5.2. Alias

5.3. Liquid types in Datay...

5.1. Specifications

5.2. Alias

5.3. Liquid types in Datay...

5.1. Specifications

5.2. Alias

5.3. Liquid types in Datay...

5.1. Specifications

5.2. Alias

5.3. Liquid types in Datay...

5.1. Specifications

5.2. Alias

5.3. Liquid types in Datay...

5.1. Specifications

5.2. Alias

5.3. Liquid types in Datay...

5.1. Specifications

5.2. Alias

5.3. Liquid types in Datay...

5.1. Specifications

5.2. Alias

5.3. Liquid types in Datay...

5.1. Specifications

5.2. Alias

5.3. Liquid types in Datay...

5.1. Specifications

5.2. Alias

5.3. Liquid types in Datay...

5.1. Specifications

5.2. Alias

5.3. Liquid types in Datay...

5.1. Specifications

5.2. Alias

5.3. Liquid types in Datay...

5.1. Specifications

5.2. Alias

5.3. Liquid types in Datay...

5.1. Specifications

5.2. Alias

5.3. Liquid types in Datay...

5.1. Specifications

5.2. Alias

5.3. Liquid types in Datay...

5.1. Specifications

5.2. Alias

5.3. Liquid types in Datay...

5.1. Specifications

5.2. Alias

5.3. Liquid types in Datay...

5.1. Specifications

5.2. Alias

5.3. Liquid types in Datay...

5.1. Specifications

5.2. Alias

5.3. Liquid types in Datay...

5.1. Specifications

5.2. Alias

5.3. Liquid types in Datay...

5.1. Specifications

5.2. Alias

5.3. Liquid types in Datay...

5.1. Specifications

5.2. Alias

5.3. Liquid types in Datay...

5.1. Specifications

5.2. Alias

5.3. Liquid types in Datay...

5.1. Specifications

5.2. Alias

5.3. Liquid types in Datay...

5.1. Specifications

5.2. Alias

5.3. Liquid types in Datay...

5.1. Specifications

5.2. Alias

5.3. Liquid types in Datay...

5.1. Specifications

5.2. Alias

5.3. Liquid types in Datay...

5.1. Specifications

5.2. Alias

5.3. Liquid types in Datay...

5.1. Specifications

5.2. Alias

5.3. Liquid types in Datay...

5.1. Specifications

5.2. Alias

5.3. Liquid types in Datay...

5.1. Specifications

5.2. Alias

5.3. Liquid types in Datay...

5.1. Specifications

5.2. Alias

5.3. Liquid types in Datay...

5.1. Specifications

5.2. Alias

5.3. Liquid types in Datay...

5.1. Specifications

5.2. Alias

5.3. Liquid types in Datay...

5.1. Specifications

5.2. Alias

5.3. Liquid types in Datay...

5.1. Specifications

5.2. Alias

5.3. Liquid types in Datay...

5.1. Specifications

5.2. Alias

5.3. Liquid types in Datay...

5.1. Specifications

5.2. Alias

5.3. Liquid types in Datay...

5.1. Specifications

5.2. Alias

5.3. Liquid types in Datay...

5.1. Specifications

5.2. Alias

5.3. Liquid types in Datay...

Refined Datatypes

So far, we have seen how to refine the types of *functions*, to specify, for example, pre-conditions on the inputs, or post-conditions on the outputs. In this section we will see how to apply these in datatypes. First, by defining properties of data values, and then by defining *datatypes* that satisfy certain invariants. In the latter, it is handy to be able to directly refine the *data* definition, making it impossible to create illegal inhabitants.

Measures are used to define *properties* of Haskell data values that are useful for specification and verification.

A **measure** is a *total* Haskell function, 1. With a *single* equation per data constructor, and 2. Guaranteed to *terminate*, typically via structural recursion.

We can tell LiquidHaskell to *lift* a function meeting the above requirements into the refinement logic by declaring:

```
(-# measure nameOfMeasure #-)
```

For example, for a list we can define a way to *measure* its size with the following function.

```
(-# measure size #-)
(-# size :: [a] -> Nat #-)
size :: [a] -> Int
size [] = 0
size (_,rs) = 1 + size rs
```

Then, we can use this measure to define aliases.

Let's create another measure named *notEmpty* that takes a list as input and returns a *Bool* with the information if it is empty or not.

```
(-# write notEmpty measure
```

Answer

We can now define a couple of useful aliases for describing lists of a given dimension.

For example, we can define that a list has exactly *N* elements.

```
(-# type ListN a N = {v:[a] | size v == N} #-)
```

Note that when defining refinement type aliases, we use uppercase variables like *N* to distinguish *value* parameters from the lowercase *type* parameters like *a*.

Now, try to create an alias *NEList* for a non-empty list, using the measure *notEmpty* created before. When removed from comment, the first example should raise an error while the second should not.

```
(-# write the alias here

-- Remove the comments below to test the alias
-- (-# ne2 :: NEList Int #-)
-- ne2 = [1,2,4] :: Int[] -- accept
-- (-# ne1 :: NEList Int #-)
-- ne1 = [] :: Int[] -- reject
```

Answer

Sparse Vectors

As our first example of a refined datatype, let's see Sparse Vectors. While the standard Vector is great for dense arrays, often we have to manipulate sparse vectors where most elements are just 0. We might represent such vectors as a list of index-value tuples (*[Int, a]*).

Let's create a new datatype to represent such vectors:

```
(-# data Sparse a = SP { spDim :: Int
                      , spTups :: [(Int, a)] } #-)
```

Thus, a sparse vector is a pair of a dimension and a list of index-value tuples. Implicitly, all indices *other* than those in the list have the value 0 or the equivalent value type *a*.

Sparse vectors satisfy two crucial properties. 1. the dimension stored in *spDim* is non-negative;

2. every index in *spTups* must be valid, i.e. between 0 and the dimension.

Unfortunately, Haskell's type system does not make it easy to ensure that *illegal vectors* are *not* representable.

Data Invariants LiquidHaskell lets us enforce these invariants with a refined data definition:

```
(-# data Sparse a = SP ( spDim :: Nat
                      , spTups :: [(BtwN 0 spDim, a)] ) #-)
```

Where, as before, we use the aliases:

```
(-# type Nat      = {v:Int | 0 <= v} #-)
(-# type BtwN Lo Hi = {v:Int | Lo <= v && v < Hi} #-)
```

Refined Data Constructors The refined data definition is internally converted into refined types for the data constructor *SP*. So, by using refined input types for *SP* we have automatically converted it into a *smart* constructor that ensures that every instance of a *Sparse* is legal. Consequently, LiquidHaskell verifies:

```
okSP :: Sparse String
okSP = SP 5 [ (0, "cat")
            , (3, "dog") ]
```

but rejects, due to the invalid index:

```
badSP :: Sparse String
badSP = SP 5 [ (8, "cat")
            , (6, "dog") ]
```

Write another example of a Sparse data type that is *illegal*. Remove the comment from the type signature below and complete the implementation with the example.

```
(-# badSP' :: Sparse String
```

Answer

Field Measures It is convenient to write an alias for sparse vectors of a given size *N*. So that we can easily say in a refinement that we have a sparse vector of a certain size.

For this we can use *measures*.

Similarly, the sparse vector also has a *measure* for its dimension, and it is already defined by *spDim*, so we can use it to create the new alias of sparse vectors of size *N*.

Think Aloud:

For the following exercise, we will use a technique called Think Aloud, where you should try to say everything that comes to your mind while you engage with the exercise.

In specific, aim:

- to speak all thoughts, even if they are unrelated to the task;
- to refrain from explaining the thoughts;
- to not try to plan out what to say;
- to imagine that you are alone and speaking to yourself; and
- to speak continuously.

For the following exercise, read the question aloud and remember to voice your thoughts while solving the exercise.

Following what we did with the lists, write the alias *SparseN* for sparse vector of length *N*, using *spDim* instead of *size*.

```
(-# write the alias here
```

Answer

You finished the Tutorial! Tell the interviewers you got to the end of the page, and answer some questions from our team before moving to the next section.

Now that you have learned the main blocks of LiquidHaskell, let's complete an exercise using all the concepts.

You can open a [Cheat Sheet](#) with examples of the main concepts on another tab on the side.

Next

- 1.Introduction
 - 1.1. Well-Typed Programs ...
 - 1.2. Refinement Types
- 2.Refinement Types
 - 2.1. Defining Types
 - 2.2. Errors
 - 2.3. Subtyping
 - 2.4. Writing Specifications
 - 2.5. Refining Function Typ...
 - 2.6. Dependent Refinements
- 3.Refined Datatypes
 - 3.1. Sparse Vectors
- 4.Case Study: Okasaki's Lazy Queues
 - 4.1. Queues
 - 4.2. Sized Lists
 - 4.3. Queue Type
 - 4.4. Queue Operations
 - 4.5. Recap of everything!
- 5.Cheat Sheet
 - 5.1. Specifications
 - 5.2. Alias
 - 5.3. Liquid types in Dataty...
 - 5.4. Measures
 - 5.1. Specifications
 - 5.2. Alias
 - 5.3. Liquid types in Dataty...
 - 5.4. Measures
- 6.Refined Datatypes
 - 6.1. Sparse Vectors

Cheat Sheet

Welcome to the LiquidHaskell Short Tutorial Cheat Sheet!

Here are the main concepts and examples you can use to complete the exercises.

Specifications

LiquidHaskell specifications in functions are written between `{-@ spec @-}`.

For example:

```
{-@ calcPer :: {a:Int | a > 0} -> {b:Int | 0 <= b && b <= a} ▶-}
calcPer    :: Int -> Int -> Int
calcPer a b = (b * 100) `div` a
```

Alias

To reuse a specification we can use aliases.

For example:

```
{-@ type Nat      = {v:Int | 0 <= v} @-}
{-@ type Btwn Lo Hi = {v:Int | Lo <= v && v < Hi} @-}
```

Liquid types in Datatypes

To add a specification to the datatypes, first create the datatype in Haskell and then add the specification inside `{-@ @-}`.

For example:

1. In Haskell

```
data Sparse a = SP { spDim    :: Int
                    , spElems :: [(Int, a)] }
```

2. Adding the LiquidHaskell specification

```
{-@ data Sparse a = SP { spDim    :: Nat
                        , spElems :: [(Btwn 0 spDim, a)] } @-}
```

Measures

Measures lift an Haskell function to the refinements logic. It is first created as a Haskell function, sinalizing that it is a measure and adding liquid types to the signature. Then, it can be used inside other refinements.

For example:

```
{-@ measure mySize @-}
{-@ mySize :: [a] -> Nat @-}
mySize :: [a] -> Int
mySize [] = 0
mySize (_,rs) = 1 + mySize rs
```

And then, `size` can be used in:

```
{-@ type ListN a N = {v:[a] | size v == N} @-}
```