Refinement Types in Haskell

Lecture 1: Introduction to refinement types

Brandon Hewer

Motivation

Type systems are useful.*

*If we care about our code being correct.

Motivation

Type systems are useful.*

*If we care about our code being correct.

Motivation

Expressive type systems are more useful.

Motivation

Dependent types give rise to particularly expressive type systems whereby programs correspond to constructive mathematical proofs.

For example, we can define safe list-indexing in Idris as follows:

```
index : Fin len → Vect len elem → elem
index FZ     (x :: xs) = x
index (FS k) (x :: xs) = index k xs
```

Motivation

Dependent types give rise to particularly expressive type systems whereby programs correspond to constructive mathematical proofs.

For example, we can define safe list-indexing in Idris as follows:

```
index : Fin len → Vect len elem → elem
index FZ     (x :: xs) = x
index (FS k) (x :: xs) = index k xs
```

Motivation

However, the expressive power of dependent types comes with a variety of costs. We will primarily focus on just one of these:

It is *very* challenging to introduce dependent types to an existing language's ecosystem.

Data point: Dependent Haskell

Motivation

However, the expressive power of dependent types comes with a variety of costs. We will primarily focus on just one of these:

It is *very* challenging to introduce dependent types to an existing language's ecosystem.

Data point: Dependent Haskell

Motivation

A refinement type system equips an existing type system with an 'extra layer' of *erasable* typing.

This extra layer allows us to *refine* types to capture additional *decidable* properties, i.e. non-empty lists, integers greater than 5.

Erasure corresponds to stripping away this extra layer and forgetting the additional properties that we captured.

Motivation

A refinement type system equips an existing type system with an 'extra layer' of *erasable* typing.

This extra layer allows us to *refine* types to capture additional *decidable* properties, i.e. non-empty lists, integers greater than 5.

Erasure corresponds to stripping away this extra layer and forgetting the additional properties that we captured.

Motivation

A refinement type system equips an existing type system with an 'extra layer' of *erasable* typing.

This extra layer allows us to *refine* types to capture additional *decidable* properties, i.e. non-empty lists, integers greater than 5.

Erasure corresponds to stripping away this extra layer and forgetting the additional properties that we captured.

Motivation

By requiring decidability, refinement type systems prioritise inference and automation first and improving expressivity is an ongoing goal.

This is in contrast to dependent type systems that typically prioritise expressivity first and for which proof automation is an ongoing goal.

Course Outline

This course will provide an introduction to refinement types in Liquid Haskell.

We will both consider applications of refinement types to general-purpose functional programming and introduce the key theoretical concepts.

We will also cover some advanced techniques, including termination metrics and higher-order reasoning via abstract refinements. In the final lecture, we will introduce quotient types within the context of a refinement type system.

Course Outline

You can install Liquid Haskell locally or use the online editor at:

https://liquidhaskell.goto.ucsd.edu/index.html

Course Outline

Lecture 1: Introduction to refinement types
Lecture 2: Refinement logic, datatypes and subtyping
Lecture 3: Measures, proofs, and termination
Lecture 4: Quotient types

Lecture Outline

1. Motivation
2. Course Outline
3. **Lecture Outline**
4. What are refinement types?
5. Refinement vs subtyping: is there a difference?
6. History of refinement types
7. Refinement type systems
8. Liquid Haskell

What are refinement types?

A refinement type system is a *conservative* extension of an underlying type system that is equipped with:

1. A *refinement relation* on types:

$$T <: U \qquad\qquad "T \text{ refines } U"$$

2. A standard subtyping rule:

$$\frac{\Gamma \vdash T <: U \qquad \Gamma \vdash e : T \qquad \Gamma \vdash U}{\Gamma \vdash e : U}$$

What are refinement types?

A refinement type system is a *conservative* extension of an underlying type system that is equipped with:

1. A *refinement relation* on types:

$$T <: U \qquad\qquad "T \text{ refines } U"$$

2. A standard subtyping rule:

$$\frac{\Gamma \vdash T <: U \qquad \Gamma \vdash e : T \qquad \Gamma \vdash U}{\Gamma \vdash e : U}$$

What are refinement types?

For example, in a type system with types Int, Nat, Even and Odd:

$$\mathsf{Nat},\ \mathsf{Even},\ \mathsf{Odd} <: \mathsf{Int}$$

That is, the types Nat, Even and Odd are *refinements* of Int.

What are refinement types?

Let us consider the operation $(+) : \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int}$.

Our subtyping rule allows us to derive the following types for the expression $\lambda\, x\ y.x + y$:

$$\mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Int}, \quad \mathsf{Even} \to \mathsf{Even} \to \mathsf{Int}, \quad \mathsf{Odd} \to \mathsf{Odd} \to \mathsf{Int}.$$

We will later consider how more precise refinements can be assigned to this expression, e.g. $\mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$.

What are refinement types?

Let us consider the operation $(+) : \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int}$.

Our subtyping rule allows us to derive the following types for the expression $\lambda\, \mathsf{x}\ \mathsf{y}.\mathsf{x} + \mathsf{y}$:

$$\mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Int}, \quad \mathsf{Even} \to \mathsf{Even} \to \mathsf{Int}, \quad \mathsf{Odd} \to \mathsf{Odd} \to \mathsf{Int}.$$

We will later consider how more precise refinements can be assigned to this expression, e.g. $\mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$.

What are refinement types?

Let us consider the operation $(+) : \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int}$.

Our subtyping rule allows us to derive the following types for the expression $\lambda\, \mathsf{x}\ \mathsf{y}.\mathsf{x} + \mathsf{y}$:

$$\mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Int}, \quad \mathsf{Even} \to \mathsf{Even} \to \mathsf{Int}, \quad \mathsf{Odd} \to \mathsf{Odd} \to \mathsf{Int}.$$

We will later consider how more precise refinements can be assigned to this expression, e.g. $\mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$.

Refinement vs subtyping: is there a difference?

Type refinement is a form of subtyping that is introduced as an extension to an underlying type system and is:

- *strongly conservative*: semantics, type-checking and the equational theory of the underlying (sub)language should remain unchanged;

- *decidable*: there is an algorithm that can check whether the subtyping relation holds between any two types and is guaranteed to terminate in finite time.

Refinement vs subtyping: is there a difference?

Type refinement is a form of subtyping that is introduced as an extension to an underlying type system and is:

- *strongly conservative*: semantics, type-checking and the equational theory of the underlying (sub)language should remain unchanged;

- *decidable*: there is an algorithm that can check whether the subtyping relation holds between any two types and is guaranteed to terminate in finite time.

Refinement vs subtyping: is there a difference?

Type refinement is a form of subtyping that is introduced as an extension to an underlying type system and is:

- *strongly conservative*: semantics, type-checking and the equational theory of the underlying (sub)language should remain unchanged;

- *decidable*: there is an algorithm that can check whether the subtyping relation holds between any two types and is guaranteed to terminate in finite time.

Strong conservativity

Strong conservativity is precisely the property that ensures that refinement types can be safely integrated into a language while continuing to make use of existing libraries and tools.

Strong conservativity

Semi-formally, given two type theories T_1 and T_2, if the syntax of T_1 is a sublanguage of T_2 then we say that T_2 is strongly conservative over T_1 if it:

- *is conservative*: for any expressible judgement J in T_1 then J is derivable in T_1 if and only if it is derivable in T_2.
- *preserves reduction*: for any terms t, u in T_1 then t reduces to u in T_1 if and only if t reduces to u in T_2.
- *preserves equational theory*: for any terms t, u in T_1 then $t = u$ in T_1 if and only if $t = u$ in T_2.

Strong conservativity

Semi-formally, given two type theories T_1 and T_2, if the syntax of T_1 is a sublanguage of T_2 then we say that T_2 is strongly conservative over T_1 if it:

- *is conservative*: for any expressible judgement J in T_1 then J is derivable in T_1 if and only if it is derivable in T_2.
- *preserves reduction*: for any terms t, u in T_1 then t reduces to u in T_1 if and only if t reduces to u in T_2.
- *preserves equational theory*: for any terms t, u in T_1 then $t = u$ in T_1 if and only if $t = u$ in T_2.

Strong conservativity

Semi-formally, given two type theories T_1 and T_2, if the syntax of T_1 is a sublanguage of T_2 then we say that T_2 is strongly conservative over T_1 if it:

- *is conservative*: for any expressible judgement J in T_1 then J is derivable in T_1 if and only if it is derivable in T_2.
- *preserves reduction*: for any terms t, u in T_1 then t reduces to u in T_1 if and only if t reduces to u in T_2.
- *preserves equational theory*: for any terms t, u in T_1 then $t = u$ in T_1 if and only if $t = u$ in T_2.

Strong conservativity

Semi-formally, given two type theories T_1 and T_2, if the syntax of T_1 is a sublanguage of T_2 then we say that T_2 is strongly conservative over T_1 if it:

- *is conservative*: for any expressible judgement J in T_1 then J is derivable in T_1 if and only if it is derivable in T_2.
- *preserves reduction*: for any terms t, u in T_1 then t reduces to u in T_1 if and only if t reduces to u in T_2.
- *preserves equational theory*: for any terms t, u in T_1 then $t = u$ in T_1 if and only if $t = u$ in T_2.

History of refinement types

1991: Refinement Types for ML. T. Freeman and F. Pfenning.

Refinement types can be introduced by a rectype declaration:

$$\begin{aligned} &\mathsf{datatype}\ \mathsf{nat} = \mathsf{zero} \mid \mathsf{suc}\ \mathbf{of}\ \mathsf{nat} \\ &\mathsf{rectype}\ \mathsf{even} = \mathsf{zero} \mid \mathsf{suc}\ (\mathsf{suc}\ \mathsf{even}) \\ &\mathsf{rectype}\ \mathsf{odd}\ = \mathsf{suc}\ \mathsf{even} \end{aligned}$$

This can be viewed as ‘structural subtyping’ for inductive types.

History of refinement types

1991: Refinement Types for ML. T. Freeman and F. Pfenning.

This notion of refinement naturally admits a lattice structure on types with an *intersection* ($\wedge$) and *union* ($\vee$) operator.

For example:

$\text{even} \wedge \text{odd} = \bot$

$\text{even} \vee \text{odd} = \text{nat}$

History of refinement types

1991: Refinement Types for ML. T. Freeman and F. Pfenning.

The *principal type* of a term is given by the greatest lower bound of all valid types that it may be assigned. For constructors, this is simply calculated as the finite intersection over its given types.

History of refinement types

1997: Construction of abstract states graphs with PVS. S. Graf and H. Saïdi.

2002: Predicate abstraction for software verification. C. Flanagan and S. Qadeer.

Predicate abstraction as applied to type systems allows us to form types from logical formula with free variables.

For example, given a logical formula Q with a free variable $n : \mathbb{N}$, then elements of $\lambda n.Q$ are natural numbers for which Q holds.

History of refinement types

1997: Construction of abstract states graphs with PVS. S. Graf and H. Saïdi.

2002: Predicate abstraction for software verification. C. Flanagan and S. Qadeer.

Predicate abstraction as applied to type systems allows us to form types from logical formula with free variables.

For example, given a logical formula Q with a free variable $n : \mathbb{N}$, then elements of $\lambda n.Q$ are natural numbers for which Q holds.

History of refinement types

1997: Construction of abstract states graphs with PVS. S. Graf and H. Saïdi.

2002: Predicate abstraction for software verification. C. Flanagan and S. Qadeer.

Predicate abstraction as applied to type systems allows us to form types from logical formula with free variables.

For example, given a logical formula Q with a free variable $n : \mathbb{N}$, then elements of $\lambda n.Q$ are natural numbers for which Q holds.

History of refinement types

2006: Inference of user-defined type qualifiers and qualifier rules. B. Chin, S. Markstrum, T. D. Millstein, and J. Palsberg.

2008: Liquid Types. P. M. Rondon, M. Kawaguchi, R. Jhala.

Liquid types are a class of refinement types that are introduced by predicate abstraction over logical expressions that are constructed from a restricted, decidable set of (quantifier-free) predicates.

History of refinement types

2006: Inference of user-defined type qualifiers and qualifier rules. B. Chin, S. Markstrum, T. D. Millstein, and J. Palsberg.

2008: Liquid Types. P. M. Rondon, M. Kawaguchi, R. Jhala.

Liquid types are a class of refinement types that are introduced by predicate abstraction over logical expressions that are constructed from a restricted, decidable set of (quantifier-free) predicates.

History of refinement types

2006: Inference of user-defined type qualifiers and qualifier rules. B. Chin, S. Markstrum, T. D. Millstein, and J. Palsberg.

2008: Liquid Types. P. M. Rondon, M. Kawaguchi, R. Jhala.

A *liquid type judgement* is given by:

$$\Gamma \vdash_{\mathbb{Q}} e : \tau$$

where $\mathbb{Q}$ is a qualifier set of decidable predicates and τ is a refinement type of the form $\{x : \sigma \mid P\}$. P is a predicate over σ that can be constructed from $\mathbb{Q}$.

History of refinement types

2015: Functors are Type Refinement Systems. P. Melliès and N. Zeilberger.

Erasure of type refinements can be modelled as a functor and this presents a category-theoretic perspective of refinement type systems.

That is, a type refinement system is a functor $F : R \to T$ where T is the underlying type theory and R is the refinement system.

History of refinement types

2015: Functors are Type Refinement Systems. P. Melliès and N. Zeilberger.

Erasure of type refinements can be modelled as a functor and this presents a category-theoretic perspective of refinement type systems.

That is, a type refinement system is a functor $F : R \to T$ where T is the underlying type theory and R is the refinement system.

Refinement type systems

Refinement type systems have been implemented for a variety of languages including:

OCaML: Dsolve: Safety Verification via Liquid Types. M. Kawaguchi, P. M. Rondon, and R. Jhala. 2010.

C: Low-level liquid types. Patrick M Rondon, Ming Kawaguchi, and Ranjit Jhala. 2010.

Typescript: Refinement types for TypeScript. P. Vekris, B. Cosman, and R. Jhala. 2016.

Liquid Haskell

Liquid Haskell adds liquid types to Haskell and discharges proof obligations to an external SMT solver.

Based on the original work on Liquid Types, and has a decidable expression language for predicates.

Liquid Haskell

Type refinements must be defined within a special comment block and are processed after standard type-checking, e.g.

$$\{-@\ \text{type Nat} = \{\text{n} : \text{Int} \mid \text{n} > 0\}\ @-\}$$

A refinement type in Liquid Haskell is introduced with a *typed binding* and a *predicate* that is constructed from the restricted logical expression language.

Liquid Haskell

The types of function definitions can similarly be refined within special comment blocks. For example:

```
{-@ church :: Nat → (a → a) → a → a @-}
church 0 f = id
church n f = church (n - 1) f . f
```

The above type definition for church will ensure it can only be applied to integers greater than or equal to 0 for which it will provably terminate.

Liquid Haskell

Example of code that will be rejected with a compile-time error by Liquid Haskell:

Example 1

```
{−@ mult :: Nat → Int → Nat @−}
mult m n = m * n
```

Example 2

```
{−@ type NonEmpty a = { xs:[a] | len xs > 0 } @−}

{−@ rest :: NonEmpty a → NonEmpty a @−}
rest (x : xs) = xs
```

Liquid Haskell

Example of code that will be rejected with a compile-time error by Liquid Haskell:

Example 1

```
{-@ mult :: Nat → Int → Nat @-}
mult m n = m * n
```

Example 2

```
{-@ type NonEmpty a = { xs:[a] | len xs > 0 } @-}

{-@ rest :: NonEmpty a → NonEmpty a @-}
rest (x : xs) = xs
```

Next Lecture

In the next lecture we will look more closely at the refinement logic of Liquid Haskell and its subtyping rules. We'll consider a number of practical examples to illustrate how refinement types can be applied.

We will also consider refinements of both inductive algebraic datatypes and record types.

Please join me in the labs to experience programming with refinement types in Liquid Haskell yourself.