Tiny Idris Program Synthesis

Scott Mora

01-2021

Structure

Introduce the structure of functional programs

Introduce the project.

Look at existing systems.

Look at the current progress.

Introduce the testing strategy and future development.

Short demo

Functional Programming

Uses the structure of data to proceed using pattern matching definitions.

Lends itself to immutable data structures, limiting code with unintended side effects.

Functions can be composed allowing for more concise code.

Data Type Declarations

Booleans

```
data Bool : Type where
  True : Bool
  False : Bool
```

Lists

```
data IntegerList : Type where
   Nil : IntegerList
   Cons : Integer -> IntegerList -> IntegerList
```

List example

```
1 :: (2 :: (3 :: []))
```

Type Signatures & Function Definitions

Type Signature

```
isEmpty : IntegerList -> Bool
```

Function Definition

```
isEmpty : IntegerList -> Bool
isEmpty Nil        = True
isEmpty Cons i is = False
```

Polymorphism

Polymorphic Lists

```
data List : Type -> Type where
   Nil  : List a
   Cons : a -> List a -> List a
```

Lists

```
bList : List Boolean
bList = True :: False :: True :: []

iList : List Integer
iList = 1 :: 2 :: 3 :: []
```

Polymorphic isEmpty

```
isEmpty : List a -> Bool
isEmpty Nil         = True
isEmpty Cons i is = False
```

Dependent Types

Numbers

```
data Num : Type where
  Zero : Num
  OnePlus : Num -> Num
```

Vect

```
data Vect : Num -> Type -> Type where
   Nil  : Vect Zero a
   Cons : a -> Vect n a -> Vect (OnePlus n) a
```

Specification

Similar structures can lead to repetitive code.

Boilerplate can lead to human error.

Tiny Idris is a dependently typed functional language.

Extend the language with program synthesis capabilities via holes.

Hole Driven Development

```
append : (a : _) -> (n : _) -> (m : _) ->
         Vect n a -> Vect m a ->
         Vect (add n m) a
pat a : Type, m : Num, ys : Vect m a =>
  append a Zero m (Nil a) ys = ?v01
pat a : Type, n : Num, x : a, xs : Vect n a,
    m : Num, ys : Vect m a =>
  append a (OnePlus n) m (Cons a n x xs) ys
    = ?v02
```

Specialised Systems

Specialised tools such as Leon, Myth, Synquid, ReSyn.

Began having type information available, but without using it.

Relied on input output examples, and other specifications.

Progressively use of type information increased.

Greatly improved performance when enough type information is available.

Struggles with less specification, eg. Lists.

All require verbose specifications.

Built in to Languages

Developed from automatic proof search tools.

Can be general or more specialised.

Idris has a more general approcach.

Agda and Coq have more hardcoded information using tactics.

Packaged in a more useful manner than synthesis systems.

Perform better than earlier systems.

Trade offs are made with more advanced systems such as Synquid.

Implementation

Identified holes during parsing along with required information.

'RawImp' holes are elabourated to the core representation.

Uses built in unification to check if guesses are valid.

Checks if any local variables are suitable.

Checks if any data constructors will result in a valid term, and attempts to synthesise arguments.

Checks if any functions would result in a valid term and synthesises arguments if so.

Each returns a list of candidates which can be sorted based on some heuriustic.

Unelabourates to RawImp and re-sugars to a string of language syntax that may be inserted.

Testing

Tests have been divided by common structures, each serving a purpose.

Lists, Vectors, Equality, Sorting and AVL trees.

Vectors are the simplest.

Lists test lack of type information.

Equality compares to more specialised proof search systems.

Sorting looks at case splitting and use of predicates.

AVL are considerably larger.

Synthesis can be called individually or in batches based on files.

Two files partially implemented, currently improvements are required before more tests are.

Future

Improve base functionallity.

Finalise the test suite.

Introduce case splitting definitions.

Introduce better heuriustics for choosing an acceptable solution.

Evaluate the performance at each step.