Tiny Idris Program Synthesis

Scott Mora

01-2021

Motivation

Views the structure of data to determine how to proceed.

Results in concise code that encourages reuse.

However, similar structures can lead to repetitive code.

Boilerplate can lead to human error.

Data Types & Function Definitions

Data Types

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

Instantiated Lists

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

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

Function Definition

```
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
```

Vect

```
append : Vect n a -> Vect m a -> Vect (n + m) a
append []        ys = ys
append (x :: xs) ys = x :: (append xs ys)
```

Specification

Tiny Idris is a dependently typed functional language.

We aim to extend the language with program synthesis capabilities via holes.

Hole Driven Development in Tiny Idris

```
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, but limited approach.

Agda and Coq have more hardcoded information using tactics languages.

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.

Tiny Idris Implementation

Syntax is desugared into RawImp, the high level representation.

RawImp is elabourated to TT, the core language used internally.

TT expressions can be evaluated to values.

Unification checks a sort of equality between terms.

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, then for function definitions.

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.

Future

Improve base functionallity.

Finalise the test suite.

Introduce pattern matching definitions.

Introduce better heuriustics for choosing an acceptable solution.

Evaluate the performance at each step.