

Tiny Idris Program Synthesis

Scott Mora

03-2021

Introduction

Introduce Functional programs and the TinyIdris system.

Project aim

Related work

Implementation

Evaluation



Defines the structure of data.

Functions transform data by inspecting the structure, and deciding how to proceed.

Abstractions lead to concise code and encourages reuse.

Similar structures can lead to repetitive code.

Repetitive boilerplate can be tedious and lead to simple errors.

Tiny Idris is a dependently typed functional language.

Not compiled, runs as an evaluator.

```
Processed Nat
processed type add
Processed add
> add (Suc (Suc Zero)) (Suc Zero)
Checked: (add (Suc (Suc Zero)) (Suc Zero))
Type: Nat
Evaluated: (Suc (Suc (Suc Zero)))
> █
```

Extend the TinyIdris system with program synthesis functionality.

Constructing a function definition from a type signature, local environment and global context.



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

No use of available type information initially, use general constraint solving approach.

Utilised specifications such as input / output examples and counterexamples.

Extend languages with constraints and type refinements.

Greatly improved performance when enough type information is available.

Can require verbose specifications.



Developed from automatic proof search tools.

Use more hardcoded knowledge.

Agda and Coq have more hard-coded information to define tactics.

Packaged in a more useful manner than specialised systems.

Idris uses a more general, though limited approach.

More focus on quick definitions using time limits.

TinyIdris Processing

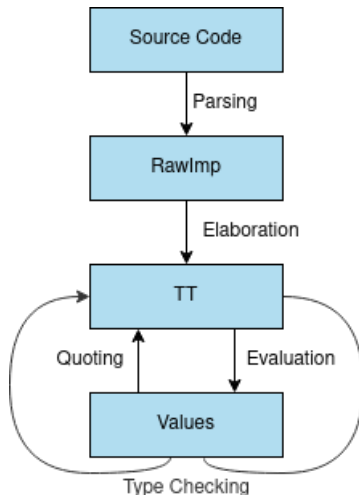
Code is parsed into the high level representation, RawImp.

RawImp is elaborated to the core language, TT.

Elaboration performs type checking and strips down terms.

Unification checks a type of equality between terms.

Resulting terms are stored in the context.



Implementation

Language extended with holes '`?hole_name`'.

Synthesise individual terms or full pattern matching definitions.

Repl extended with '`auto`' command for interaction.

Testing functionality using pre-defined answer files and '`t, test`' commands.

Attempts full context enumeration unlike other dependently typed systems.

Term Synthesis

Local environment is searched first.

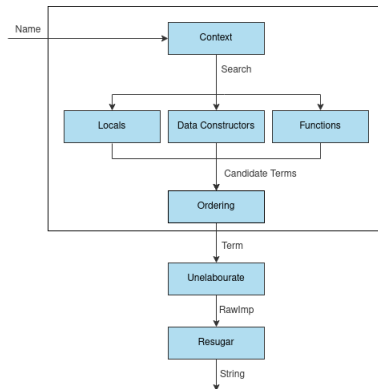
Data Constructors second.

Finally function definitions.

Arguments are synthesised in a depth first traversal.

Returns a list of results to be ordered.

Order of synthesis is a heuristic.



Definition Synthesis

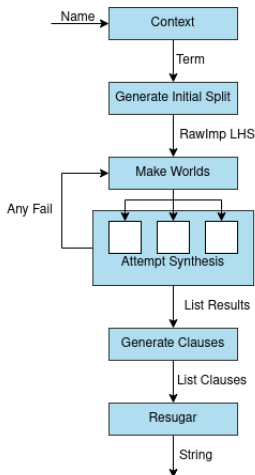
Given a type signature,
construct a left hand side term
to be split.

Alternates between splitting on
arguments and attempting
synthesis.

Synthesis attempted for each
left hand side.

If all are successful the results
are combined.

If any fail the process attempts
further splitting.



Case Splitting

An Initial split is performed on the leftmost type constructor.

Splitting on all names that depend on the split argument is attempted.

If the split results in only one case, new names are added and the process repeats until no names are left.

```
pat a : Type, n : Nat, m : Nat,  
  xs : Vec n a, ys : Vec m a =>
```

```
pat a : Type, m : Nat, ys : Vec m a =>  
  append a Z m (Nil a) ys
```

```
pat a : Type, n : Nat, m : Nat, x : a, xs : Vec n a,  
  ys : Vec m a =>  
  append a (S n) m (Cons a n x xs) ys
```

Case Splitting

LHS split into patterns before the split argument, and the scope.

Identify which patterns for the constructor have already been introduced.

Generate unique pattern variables for those which have not.

Replace references to the term with the application of the constructor.

Put the term back together and attempt to fill in any implicits generated.

```
pat a : Type, n : Nat m : Nat xs : Vec n a ys : Vec m a =>
```

```
pat a : Type, n' : Nat m : Nat xs : Vec (S n') a ys : Vec m a =>
```

Tested with varying levels of type information.

Compared to other similar systems.

Idris, Idris2, Agda, Synquid.

Evaluated based on the correctness of synthesised definitions.

On each input returns the expected output.

Evaluation - Lists

| Problem | TinyIdris | Idris | Idris2 | Agda | Synquid |
|-----------|-----------|-------|--------|------|---------|
| append | | | Pass | | |
| map | | | Pass | | |
| replicate | | | Pass | | |
| drop | | | | | |
| foldr | | | | | |
| is empty | | | | | |
| is elem | | | | | |
| duplicate | | | | | |
| zip | | | | | |
| i'th elem | | | | | |
| index | | | | | |

Evaluation - Vectors

| Problem | TinyIdris | Idris | Idris2 | Agda | Synquid |
|-----------|-----------|-------|--------|------|---------|
| append | Pass | Pass | Pass | Pass | Pass |
| map | Pass | Pass | Pass | Pass | Pass |
| replicate | Pass | Pass | | | Pass |
| drop | | | Pass | Pass | Pass |
| foldr | | | | | Pass |
| is empty | | | | | Pass |
| is elem | | | | | Pass |
| duplicate | Pass | | | | Pass |
| zip | | Pass | Pass | Pass | Pass |
| i'th elem | | | | | Pass |
| index | | | | | Pass |

Equalities

| Problem | TinyIdris | Idris | Idris2 | Agda | Synquid |
|------------------------|-----------|-------|--------|------|---------|
| andSymmetric | Pass | Pass | Pass | Pass | Pass |
| orSymmetric | Pass | Pass | Pass | Pass | Pass |
| plus commutes | | | | | - |
| plus Suc | | | | | - |
| symmetry | | Pass | Pass | Pass | |
| transitivity | | Pass | Pass | Pass | |
| congruence | | Pass | Pass | Pass | - |
| list(vec(list)) = list | | Pass | Pass | Pass | - |
| vec(list(list)) = vec | | | Pass | | - |
| disjoint union apply | | | | Pass | Pass |
| a = not not a | | | | | - |
| not not not a = not a | | | Pass | Pass | - |

Heuristics

| Problem | Loc-> DCon -> Func | Max Args | Rec + Max Arg | Functions Only |
|-----------|-----------------------|----------|------------------|-------------------|
| append | Pass | Pass | Pass | Pass |
| map | Pass | | | Pass |
| replicate | Pass | | | Pass |
| drop | | | | |
| foldr | | | | |
| is empty | | | | |
| is elem | | | | |
| duplicate | Pass | Pass | Pass | Pass |
| zip | | | | |
| i'th elem | | | | |
| index | | | | |

Effectiveness limited due to the lack of constraint solving.

Performance hindered by full enumeration.

Heuristic less effective with full enumeration of the context

Prioritising language constructs that requires more built in knowledge may perform better than context enumeration.

Type system extensions could provide more powerful definitions, as seen in Synquid.