# TinyIdris Program Synthesis

CS408 Progress report

Scott Mora - 201816428

March 10, 2021

bibliographystyle:plain

## 1 Introduction

Programming often involves implementing low level details that can be tedious. The goal of program synthesis is to automate as much of this as possible, speeding up the process of software development and reducing bugs that can be introduced through programmer error. There have been attempts to synthesise programs from many different communities. The approach considered here uses the type checker to guide the synthesis algorithm towards a solution. Say we have a List data type, and wish to synthesise a definition for the function 'map' which, when given another function, and a list, applies the function to each element of the list, a common pattern in functional programming. The list argument has been pattern matched on, and the holes to be synthesised have been added.[1]

```
data List : Type -> Type where
  Nil  : List a
  (::) : {a : Type} -> a -> List a -> List a

map : (f : a -> b) -> (xs : List a) -> List b
map f [] = ?map_rhs_1
map f (x :: xs) = ?map_rhs_1
```

Filling in the first hole is straightforward, since the expected output is the `Nil` constructor. The second, however, is more interesting. There are now multiple outputs that would satisfy the type checker. The desired output applies `f` to `x` and concatenates the result to the recursive call using the `::` constructor. However, inserting the term `[]` into the hole would also satisfy the type checker, indeed, with previous systems implemented, this is exactly the synthesised value.

By providing the type checker with more information we are able to eliminate more incorrect answers, reducing the number of incorrect answers returned, and increasing the speed by reducing the number of branches to be checked. By allowing types to depend on terms we are able to do this, as the following example displays.

```
data Vector : Nat -> Type -> Type where
  Nil  : Vector 0 a
  (::) : {a : Type} -> a -> Vector n a -> Vector (1 + n) a

map : (f : a -> b) -> Vector n a -> Vector n b
map f [] = []
map f (x :: y) = f x :: map f y
```

Here, Vector describes a family of types, where 'Lists' are indexed by their length. The first hole remains unchanged, since there is only one term satisfying `Vector 0 b`, namely, `Nil`. In the (::) case however, the empty vector is no longer valid, since the type `Vector 0 b` $\neq$ `Vector (1 + n) b`.

TinyIdris is a small dependently typed programming language. Users interact with the Tiny Idris system by loading their source files into the TinyIdris Repl (Read Eval Print Loop). Our aim is to extend

[1] The examples on this page are written in the programming language Idris. They are not valid in Idris2, the language used throughout the rest of this paper. This decision removes some added complexity from the examples. See cite:BibEntry2020Nov.

TinyIdris with program synthesis functionality. An `auto` command has been added to the repl, when given the name of a hole, attempts synthesis, returning the string representation of the synthesised term if successful, or a failure message if not.

1. TODO: why data types ext? why functional

# 2 The TinyIdris System

TinyIdris is a scaled down version of the Idris2 programming language. The system operates by calling the tinyidris executable with the filename of a tinyidris source code file '.tidr'. The file is then processed by the system, where the source code is converted into the internal representation of the language and type checking occurs, and each definition is stored in memory. A read eval print loop, or repl is then presented, which accepts expressions from the language, evaluates them, and prints out the result.

## 2.1 The Source code

Tiny Idris source code files consist of three top level constructs. Data Types, Type Signatures, and Function Definitions.

### 2.1.1 Type Signatures

Type signatures outline the shape of the data that is being transformed, and are written `f : A`, read "f has type A". Where the construct being declared by the type signature takes arguments, it is written `f : A -> B -> C`, everything to the left of the final arrow is the type of each argument, be it datatypes, or function types , and the right hand side is the return type. There can be 0 or more arguments, however there must be exactly one return type, if multiple values are returned this is done so by constructing a tuple. `f : A -> B -> (C, D, E)`. Type signatures are present in both of the following constructs.

### 2.1.2 Data Types

Data types describe how to construct values of the type being defined. They consist of a type declaration and a set of data declarations. The type declaration begins with the `data` keyword and is followed by a type signature for the type being defined. The data constructors are type signatures defined in the where block that follows.

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

data Nat : Type where
  Zero : Nat
  Suc  : Nat -> Nat
```

Defining the natural numbers we may have either zero, or one more than another natural number, here the successor constructor requires the previous number to be passed in as an argument. We construct values of a type by calling one of the data constructors and supplying the arguments.

```
two : Nat
two = Suc (Suc Zero)
```

Constructing the number two, we call the Successor constructor twice, first with the argument Zero, and with the result of that as the second.

### 2.1.3 Function Definitions

The above example is a function definition, which has no arguments, and returns a value of type Nat. Functions have two components. A type signature, and a pattern matching defintion.

```
not : Bool -> Bool
not True  = False
not False = True

even : Nat -> Bool
even Z = True
even (S n) = not (even n)
```

Function definitions have a left hand side and a right hand side. The left hand side lists the arguments that have been passed in, and the right hand side constructs a term of the return type. Pattern matching is used to inspect the arguments that have been passed in, by inspecting which data constructor has been used to construct it. All of the arguments from the left hand side are available to be used on the right hand side, and any number can be matched on.

### 2.1.4 Parametricity

Similarly to the definition of natural numbers we can define a list of Nats inductively, by first building up from the base case, and successivly adding an Nat.

```
data NatList : Type where
  Nil : Natlist
  Cons : Nat -> NatList -> NatList
```

However not all data types have type 'Type'. The language supports polymorphism in the form of indexed types, allowing lists to be defined generically.

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

The list data type implicitly recieves the pratmeter `a : Type`, which results in the type `List a`. This allows functions to operate on lists based on their structure, without inspecting the elements themselves, allowing for more code reuse.

```
map : (a -> b) -> List a -> List b
map f []        = []
map f (x :: xs) = (f x) :: map f xs
```

### 2.1.5 Dependent Types

This polymorphism is not limited to accepting only types, as Types may also depend on values. The previous example of lists can be further extended to the vectors, generic lists of a certain length.

```
data Vector : Nat -> Type -> Type where
  Nil : Vector Zero a
  Cons : a -> Vector n a -> Vector (S n) a
```

When constructed, the type of each vector will depend on the values passed in as arguments, if the Cons constructor is used and a vector of 4 elements is passed in it will have type Vector 5 a, which is a different type to a Vector 6 a, or Vector Zero a, and so on.

1. **TODO** example functions

### 2.1.6 Differences from the Examples

For simplicity of the examples, the examples displayed have not been valid TinyIdris code. We now look to see some differences with TinyIdris code.

```
data Vector : Nat -> Type -> Type where
 Nil : Vector Z a
 Cons : a -> Vector n a -> Vector (S n) a

append : Vector n a -> Vector m a -> Vector (n + m) a
append Nil ys = ys
append (x :: xs) ys = x :: append xs ys

-----------------------------------------------------------------

data Vec : Nat -> Type -> Type where
  Nil : (a : Type) -> Vec Z a
  Cons : (a : Type) -> (n : Nat) -> a -> Vec n a -> Vec (S n) a

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

The initial code presented supports implicit arguments, with the a and n not being passed in explicitly, as they are in the TinyIdris code. The other difference is the patterns being matched must be passed in explicitly. These are required due to the lack of full support for implicit arguments.

## 2.2 The Raw Implementation

Parsing results in a list of declarations. Declarations can be type signatures, ImpTy, consisting of a name and a term. Data type declarations, consisting of a term for the type, and a list of terms for the constructors. Function definitions consist of a name, and a list of clauses, where each clause has a left hand side.

```
public export
data ImpTy : Type where
   MkImpTy : (n : Name) -> (ty : RawImp) -> ImpTy

public export
data ImpClause : Type where
   PatClause : (lhs : RawImp) -> (rhs : RawImp) -> ImpClause

public export
data ImpData : Type where
   MkImpData : (n : Name) ->
         (tycon : RawImp) -> -- type constructor type
         (datacons : List ImpTy) -> -- constructor type declarations
         ImpData

public export
data ImpDecl : Type where
   IClaim : ImpTy -> ImpDecl
   IData : ImpData -> ImpDecl
   IDef : Name -> List ImpClause -> ImpDecl
```

```
public export
data RawImp : Type where
   IVar : Name -> RawImp
   IPi : PiInfo -> Maybe Name ->
       (argTy : RawImp) -> (retTy : RawImp) -> RawImp
   ILam : PiInfo -> Maybe Name ->
      (argTy : RawImp) -> (scope : RawImp) -> RawImp
   IPatvar : Name -> (ty : RawImp) -> (scope : RawImp) -> RawImp
   IApp : RawImp -> RawImp -> RawImp
   Implicit : RawImp
   IType : RawImp
```

IVars are the lowest level of terms, each term that is in scope will be referenced by an IVar expression. IPi represents function types, taking in an argument, possibly with a name, and scopem which is the rest of the term. IPatVars are found in each clause of a pattern matching definition taking in a name, term and the scope. IApp represents an application of a function to an argumemt, Type represents the type of types.

TinyIdris has limited support of implicit arguments of the form `(x :  _)` , this is represented by `Implicit`.

ILam represents anonymous functions, that take an argumemt, and returns the scope, they are represented in the language as `\ x => scope`, and would be used on the right hand side of a clause when constructing a function.

## 2.3 The Core Representation

After the source has been parsed into a list of declaration, each declaration is then processed by the elabourator. Elabouration takes in a rawimp term performs type checking to ensure that it is well typed, if a terms are found to be ill typed then processing is stopped and an error is thrown, otherwise, if each rawimp term in a declaration is valid then it is added to the context.

The context is a sorted map of names and global definitions, where each globaldef has a type and a definition.

Along with each of the definitions discussed earlier, they can also be None, which id the definition of a type signature without an accompanying function declaration, a hole, or a guess, which are used during unification.

```
public export
data Term : List Name -> Type where
   Local : (idx : Nat) -> -- de Bruijn index
       (0 p : IsVar name idx vars) -> -- proof that index is valid
       Term vars
   Ref : NameType -> Name -> Term vars -- a reference to a global name
   Meta : Name -> List (Term vars) -> Term vars
   Bind : (x : Name) -> -- any binder, e.g. lambda or pi
      Binder (Term vars) ->
      (scope : Term (x :: vars)) -> -- one more name in scope
      Term vars
   App : Term vars -> Term vars -> Term vars -- function application
   TType : Term vars
   Erased : Term vars
```

Terms can reference names from either a global or a local scope. Names that are referenced locally via a de bruijn index, and a proof that that name is at the index is valid. This enforces terms to be well scoped. Terms that are referenced globally `Ref` contain the name, along with the the nametype, which can be a type constructor, data constructor, bound variable or function.

There are four kinds of binders, Pi and PVTy both descrive the types of terms being taken in, while Lam and PVar describe the terms being taken in. For convenience have been combined into the binder type, which issue taken in by a bind term, along with a name and a scope, where a name was not provided by the user, or a term has been constructed by the system, a machine generated name is used.

Meta terms are constructed during unification, they have a name and a list of arguments to which they are applied. App and TType are equivalent to their RawImp counterparts and Erased represents erased terms.

Global terms are stored in the context, however we also require a local context, this is done via the Environment.

```
data Env : (tm : List Name -> Type) -> List Name -> Type where
   Nil : Env tm []
   (::) : Binder (tm vars) -> Env tm vars -> Env tm (x :: vars)
```

Environments are of the familiar list structure, for generality the first parameter Env takes is of type `List Name -> Type`, thus it could be an environment of any type that is indexed by a list of names, however, it will, for our purposes only ever be an environment of terms. Since the system contains dependent types, terms may reference those previously brought into scope, the second argument is a list of names, this enforces that if a term does reference an earlier term, then it is in the environment. The data constructors ensure that if the environment is empty, then there are no names in scope that can be references, each time a binder is added to the environment, then a name that may be referenced is added to the environment along with.

## 2.4 Values

Values within the TinyIdris system are in Head-Normal form, and similarly to terms, are also scoped by a list of names via the NF data type, along with some auxillary data types.

```
-- The head of a value: things you can apply arguments to
public export
data NHead : List Name -> Type where
   NLocal : (idx : Nat) -> (0 p : IsVar name idx vars) ->
        NHead vars
   NRef   : NameType -> Name -> NHead vars
   NMeta  : Name -> List (Closure vars) -> NHead vars

-- Values themselves. 'Closure' is an unevaluated thunk, which means
-- we can wait until necessary to reduce constructor arguments
public export
data NF : List Name -> Type where
   NBind    : (x : Name) -> Binder (NF vars) ->
        (Defs -> Closure vars -> Core (NF vars)) -> NF vars
   NApp     : NHead vars -> List (Closure vars) -> NF vars
   NDCon    : Name -> (tag : Int) -> (arity : Nat) ->
        List (Closure vars) -> NF vars
   NTCon    : Name -> (tag : Int) -> (arity : Nat) ->
        List (Closure vars) -> NF vars
   NType    : NF vars
   NErased  : NF vars
```

## 2.5 Processing Example

To see exactly what is happening, we shall look at some examples of processing definitions. We begin by looking at elaboration, which is used to convert RawImp to Terms, and is used when processing terms. And then the extra steps taken when processing each type of definition.

1. Elaboration

```
export
checkTerm : {vars : _} ->
      {auto c : Ref Ctxt Defs} ->
      {auto u : Ref UST UState} ->
      Env Term vars -> RawImp -> Maybe (Glued vars) ->
      Core (Term vars, Glued vars)

checkExp : {vars : _} ->
       {auto c : Ref Ctxt Defs} ->
       {auto u : Ref UST UState} ->
       Env Term vars ->
       (term : Term vars) ->
       (got : Glued vars) ->
       (expected : Maybe (Glued vars)) ->
       Core (Term vars, Glued vars)
```

Elaboration is implemented via two functions, checkTerm and checkExp (for checkExpected). Checking a term, when provided with an environment, RawImp expression, and possibly a type that we think it is, will, assuming success, return the checked Term in the core language, with its type.

Glued variables are simply terms, paired with their normal form. The checkTerm function, elaboration proceeds by breaking down terms into their individual componenents, checking each component, putting them back together and calling the checkExp function on them, for example, when checking the term `(IApp f a)`, the function is checked, and if it has the type of a Pi binder, then the argument type is checked, and check exp is called with the an `App` term with the checked function to the checked argument, and the scope of the function, after being provided the argument as the type, and the expected type provided. If the term is a Pi binder then term being taken in is checked, and the environment is extended with the term, the scope of the binder is then checked in the updated environment, checkExp is then called with the Bind term, TType as the type, and the expected value. If the given term is an `IVar` then it is checked if the name is in the local or global scope, and the resulting term that is passed to checkExp will be a `Local` or `Ref` depending on the check.

The checkExp functions purpose is to check that the term that is passed in matches the expected term, if there is no expected term then it succeeds, returning the term and its type, otherwise it attempts to unify the type of the term we have, and the type of the expected term, returning the result, otherwise failing with an error.

2. Unification Unification is the process of checking a kind of equality between two terms,

```
unify : {vars : _} ->
    {auto c : Ref Ctxt Defs} ->
    {auto u : Ref UST UState} ->
    Env Term vars ->
    tm vars -> tm vars ->
    Core UnifyResult
```

3. Processing

   The Type case is straightforward, in the empty environment, the RawImp term is checked, and the resulting term is added to the context with the definition None.

   The

## 3 Related work

There is a strong relationship between type guided program synthesis and the creation of automatic proof search algorithms. It is worth noting that there have also been attempts at synthesising programs from the machine learning community, however these are outside the scope of the project and as such are not discussed here. Some of the research presented here has since been improved with the introduction of quantitative types[2], where values are annotated with a multiplicity, stating how many times it may be used, this has been shown cite:10.1145/3314221.3314602 to improve the performance of synthesis algorithms within a type driven approach. TinyIdris does not support quantitative types, and hence these are omitted.

## 3.1 Automated Theorem Proving in Agda

Agda is a dependently typed programming language and interactive proof assistant, and is the closest relative to Idris. Indeed the development of Agda heavily influenced that of Idris cite:Splv'202020Aug . The language supports many of the same features as Idris, such as hole driven development with interactive typing information. Agsy is a tool developed and currently implemented as part of the Agda interactive development system. The user can invoke the tool via Agda while the cursor is placed within a hole, alternatively, it exists as a stand alone tool. Agsy has been developed as a proof search tool. Both the input and output (where successful) are terms in the Agda language. Agsy uses Agda's type checker, along with an extended unification algorithm to reduce the search space, however it does not propagate constraints through the search, and instead uses 'tactics' which are invoked based on the shape of the goal. Use of the built in type checker adds the requirement that Agsy must implement termination checking manually on the terms it generates, since this is not implemented within the type checker. Meta-variables are refined via a depth first traversal of the search space, and are separated into two categories, *parameter meta-variables*, and *proof meta-variables*. Only proof meta-variables require synthesised, since parameter meta-variables will be instantiated later. Eliminating a proof term occurs by searching the context, and enumerating all valid terms that result from function application, record projection or case splitting on inductive data types.

To avoid nontermination, the search uses iterative deepening, this has the added benefit that commonly, the more desirable solutions are encountered first. A problem in Agsy contains:

- A collection of parameter meta-variables, each containing a context and type
- The current instantiations for parameter meta-variables
- The context of the current problem
- The sequence of conditions that have occurred so far
- A target type

A solution is represented as a set of meta-variable instantiations, a set of conditions, and a term that inhabits the target type. Agsy also has an intermediate structure for refinements that outlines how a problem can be refined into a new set of problems, of the same form as a solution, except the term has meta-variables that are split into a set of parameter meta-variables and a set of proof meta-variables.

The tactics outlined in the paper consist of, solving equality proofs by using knowledge of congruence and reflexivity, performing induction on the parameter meta-variables to refine the goal type, case splitting on the result of evaluating an expression, and a tactic 'generalise', that either replaces multiple occurrences of a meta-variable with two different variables, or picks a sub-expression and replaces it with a new variable.

The search begins by generating a list of refinements via the tactics, then, for each refinement, attempting to solve it by searching for a term, and combining the parameter instantiations to generate the top level term. For each solution returned the algorithm attempts to lift the instantiations and refinements into the current scope, by removing bindings generated, and checking that the conditions are valid in the top level context. Accepted solutions are compared via subset inclusion of their parameter instantiations, and the best solution is returned. The conditions of generated solutions are also checked against the conditions of the already generated solutions; if successful, they are merged with the case expression to one single solution.

The result of this research is a tool which is useful for solving certain, relatively small synthesis problems, and is efficient enough to be included, and useful within Agda's interactive editing environment. One issue that the tool is hindered by is Agda's lack of a core language, this results in the tool not working for new features. Having a small core language, with a higher level implementation that is elaborated down to the core language, would allow the tool to operate only on the core language, and hence work with new language features. The tool focuses on using tactics rather than a more general approach, this

[2] Also referred to as resource types.

does mean it is limited by the expressiveness of the tactic language. However this may also work in Agsy's favour, as more general approaches may not be as effective at synthesising solutions that require specific knowledge of the problem domain.

## 3.2 Applications of Applicative Proof Search

This work constructs a library for typed proof search procedures. The approach taken is very general, which allows the framework to be easily specialised to various concrete use cases. The two examples provided use the library to implement a property based testing library and a basic model checker. Here, we cover the main framework, as it shares some similarities with synthesising definitions.

Decidability is often used as a replacement for the boolean type in dependently typed languages. To construct a value of type `Dec` there are two constructors, `yes` and `no` which take a proof of truth, or falsity respectively. *'deciders'* can be defined, for example, `_leq-dec_:(m:Nat) -> (n:Nat) -> Dec (m leq n)` is a decider for the ordering of natural numbers. Problems set in with this approach however, when properties are fundamentally undecidable. The paper, as a result defines a *hemidecider* type, `HDec` which can be constructed by `success` which takes in a proof of truth, or `failure` which requires no proof of falsity, stating that no proof was found, as opposed to no proof exists. Here we can see a clear similarity with the problem of program synthesis, as our search may return a value, or fail. The paper constructs instances of *Alternative*, *Monad* and *Applicative*, for hemideciders. The alternative instance combines two hemideciders for the same proposition by attempting one then the other. The monad instance allows results to be chained together, with each building on the result of those which came before. The applicative instance allows the application of a proof search procedure to each sub-goal more succinctly. Functions `Any` and `All` are created that, when applied to a search, a list of *X's*, and a function that when given an `X` returns a hemidecider for the given search, applies the function to each element of the list and returns success if any of the values succeeded, or all of the values succeeded, respectively. The paper goes on to develop two validation libraries. The flexibility of the framework, and similarity in the initial problem suggests that following a similar approach will provide a solid foundation for a proof search algorithm.

## 3.3 Synthesis Modulo Recursive Functions

One of the earlier systems for synthesising programs within a functional programming environment was included in the Leon system. The system is implemented in, and able to synthesise, a subset of Scala. The tool is available as both a command line tool and a web based application. Although the Synthesiser has typing information available to it, it is not used to guide the algorithm, instead it uses examples, and counterexamples to guide synthesis. Leon is a verifier that detects errors within functional programs and reports counterexamples. The system interleaves automated and manual development steps where the developer partially writes a function and leaves the rest to the synthesiser, alternatively the synthesiser may leave open goals for the programmer. This allows the user to interrupt the system at any point and get a best effort definition. The system aims to synthesise functions that manipulate algebraic data types and unbounded integers. The Synthesiser uses 'symbolic descriptions' and can accept input/output examples, in conjunction with synthesis rules that decompose problems into sub-problems. An example problem of splitting a list in the Leon system:

```
def split(lst : List) : (List , List) = choose { (r : (List , List)) =>
  content(lst) == content(r,_1) ++ content(r,_2)
}
```

This definition will synthesise an incorrect solution, however specifications can be refined by the programmer and indeed we can synthesise the correct solution:

```
def split(lst : List) : (List , List) = choose { (r : (List , List)) =>
  content(lst) == content(r,_1) ++ content(r,_2)
  && abs(size(r,_1) - size(r,_2)) <= 1
  && (size(r,_1) + size(r,_2)) == size(lst)
}
```

Internally, a synthesis problem is represented as a set of input variables, a set of output variables, a synthesis predicate, and a path condition to the synthesis problem. A path condition is a property of the inputs that must hold for synthesis is performed. The system uses a set of inference rules which outline how to decompose a term being synthesised into a simpler problem. These involve *generic reductions* which synthesise the right hand side of an assignment and outputs the assignment, *conditionals* where the output is an `if then else` statement, and can be used when the predicate contains a disjunction. *Recursion schemas* produce recursive functions and *terminal rules* generate no sub-goals. Two algorithms are then presented for computing a term given a path condition and synthesise predicate. The *Symbolic Term Exploration Rule* and the *Condition Abduction Rule*. The search alternates between considering the application of rules to given problems, and which sub-problems are generated by rule instantiations. This is modelled as an AND/OR tree.

The symbolic term exploration rule enumerates terms and prunes them using counterexamples and test cases until either a valid term has been found, or all terms have been discarded. This enumeration focuses on constructors and calls to existing functions. The problem is encoded as a set of *Recursive generators*, which are simply programs that return arbitrary values of the given type; this is converted into an SMT term which is passed into a *refinement loop*. Refinement loops search for values satisfying the condition where the synthesis predicate is true, this is restricted via iterated deepening. If a candidate program is found then it is put through another refinement loop, this time looking for inputs where the synthesis predicate does not hold in conjunction with the given formula.

There exists an alternative to this process by way of concrete examples, the Leon system generates inputs based on the path condition, and tests the candidate programs on these inputs, if a program fails on any input it may be discarded.

The condition abduction rule, when given a function signature and post condition attempts to synthesise a recursive well typed and valid, function body. This is done via searching the definitions available in the context and using condition abduction. Condition abduction is based on abductive reasoning, which seeks to find a hypothesis that explains the observed evidence in the best way. It works on the principle that recursive functional programs frequently start with top level case analysis and recursive calls within the branches. The algorithm first finds a candidate program, then searches for a condition that makes it correct. The algorithm that implements the idea begins with the set of all input values for which there is no condition abducted, a set of partial solutions, and a set of example models. The algorithm collects all possible expressions for the given expression and evaluated on the models, the models are an optimisation, that are checked against before the validity check. Candidates are ranked by counting the number of correct evaluations. The highest ranked candidate is checked for validity, if it is accepted it is returned, otherwise the counterexample is added to the models and the branching is attempted with the candidate expression. If the branching algorithm returns a result, the inputs left and solutions are updated and. This is repeated until the collection of expressions is empty.

The branching algorithm gets a set of candidates and for each checks if it can find a valid condition, it is checked against the set of models. If it prevents all counterexamples then the candidate is checked for validity, if valid the candidate is returned, otherwise the counterexample is added to the list of models.

The system was evaluated on a small set of examples, of which it managed to synthesise the majority. More recent work has surpassed it by synthesising significantly more problems, and in much less time, however techniques outlined here, such as condition abduction, which have heavily influenced techniques used in more modern systems.

## 3.4 Type and Example Directed Program Synthesis

The Myth system treats program synthesis as a proof search, that uses type information and concrete input/output examples to reduce the size of the search space. The system generates OCaml syntax, however it requires type signatures, differentiating it from the language. The work introduces the concept of *refinement trees* that represent constraints on the shape of the generated code. The main principle of the system is to use typing judgements that guide examples towards the leaves of derivation trees, thus dramatically pruning the search space.

Input/output example pairs are divided into 'worlds', each input/output pair exists in it's own world. This requires the internal representation of the language to be extended with partial functions to represent these worlds. To rule out synthesising redundant programs, terms must be $\beta$-reduced before being synthesised. Terms are also divided into introduction and elimination forms, where elimination forms are variables or applications. This is made explicit by the bidirectional typing system, which checks types for introduction forms, and generates types for elimination forms.

In order to ensure the system does not generate terms which do not terminate, it implements a structural recursion check, and positivity check. Due to the undecidability of function equality however, there are no checks for example consistency, thus if provided with inconsistent examples, there is no guarantee that the synthesis algorithm will terminate, for this reason the implementation contains a user defined depth limit.

Myth has rules for both type checking and synthesis, they are very similar, however have inverted purposes, type checking rules produce a type given a term, whereas synthesis rules produce a term given a type, these rules state how to proceed based on the given input. This introduces non-determinism into the system as it is possible that multiple rules apply at once, for example the rules *IREFINE-MATCH* and *IREFINE-GUESS* both apply to base types. The system exhaustively searches all possibilities up to a user defined limit. An optimisation the system makes when enumerating potential terms is to cache results of guessing, and attempts to maximise the sharing of contexts so that terms are only ever enumerated once.

The system operates in two modes, *E-guessing* and *I-refinement*, which involve term generation and "pushing down" examples. This is implemented via a refinement tree, which captures all possible refinements that could be performed. Refinement trees consist of two types of nodes, *Goal nodes* representing places where E-guessing can take place, and *Refinement nodes*, where I-refinement may take place. When using refinement trees the evaluation strategy consists of creating a refinement tree from the current goal and context, perform E-guessing at each node, push successful E-guesses back up the tree to try and construct a program that meets the top level criteria.

Refining via the matching rule may potentially be wasteful, since there is no guarantee that splitting on an input will provide useful information, for this reason the system implements a check to make sure that it will help progression towards a goal.

Myth was tested on a set of problems surrounding the data structures, booleans, natural numbers, lists, and trees. In the majority of cases it was able to synthesise the expected definition. In some cases it synthesised correct, however surprising results, which when looked into were slightly more efficient than the standard definitions. The tests were run both with a minimal context and more populated context, it was found that running with a larger context could increase run-time by 55%. In most cases the run-time is still relatively low, however some definitions took up to 22 seconds. Example sets also presented an issue, with some problems requiring up to 24 input/output examples to be synthesised, and in some cases coming up with examples which allowed a definition to be synthesised.

### 3.5 Program Synthesis from Polymorphic Refinement Types

Synquid is a type guided program synthesis system developed that uses the recent idea of liquid types to provide the type checker with more information to effectively reduce the search space. Liquid types allow programs to be specified in a more compact manner than using examples. Synquid has its own syntax, which contains fragments of both Haskell and Ocaml. The tool is available in a web interface. An example refinement can be seen in the type of:

```
replicate :: n : Nat -> x : A -> {List A | len v = n}
```

Where the return type `List A` has been refined by the condition that the length of the output, `v`, is equal to the number passed in. The type system also makes use of *abstract refinements*, which allow quantification of refinements over functions, for example, lists can be parameterised by a relation that defines an ordering between elements.

A problem in Synquid is represented as a goal refinement, along with a typing environment and a set of logical quantifiers, while a solution is a program term. The system, to cut out redundant refinements requires all terms to be in $\beta$-normal-$\eta$-long form in a similar fashion to systems which have come before. Due to the standalone nature of the system, the function being synthesised does not exist in the context when the system is invoked, thus it adds a recursive definition, weakened by the condition that it's first argument must be strictly decreasing. The system uses a technique named *liquid abduction* which is a similar strategy to that of condition abduction, outlined previously. One benefit of the approach taken here is the ability for the system to reason about complex invariants not explicitly stated within the type due to the additional structure present in the types.

Synthesis is split into three key areas, bidirectional type checking, sub-typing constraint solving, and the application of synthesis rules.

Following from previous work, terms are split into introduction and elimination terms. Elimination terms consist of variables and applications, and propagate type information up, combining properties of their components. Introduction terms do the opposite, breaking complex terms down into simpler ones. I-terms are further split into branching terms, conditionals using liquid types, function terms, abstractions and fix-points. Types are split into scalar (base types which may be refined), and dependent function types. The type checking rules are split into inference judgements and checking judgements. Inference rules state that a term `t` *generates* type `T` in an environment Γ. Checking rules state that a term `t` *checks against* a known type `T` in the environment Γ. The inference rules in the system have been strengthened allowing sub-typing constraints to be propagated back up, rather than abandoning the goal type at the inference phase. The system begins by propagating information down using the checking rules until a term to which no checking rule applies is reached. At this point the system attempts to infer the type of the term, and checks if it is a sub-type of the goal. Inspired by condition abduction from earlier work, the system uses *liquid abduction* to improve the effectiveness of enumerating conditionals. The type checking algorithm is further extended to the *local liquid type checking algorithm.* With this extension, during type checking, sub-typing constraints, horn constraints, type assignments and liquid assignments are maintained, and the program alternates between applying the rules and solving constraints.

Constraint solving consists of either applying a substitution, attempting unification, or decomposing sub-typing constraints and calling the horn solver. Horn constraints are of the form $\phi \Rightarrow \psi$ where $\phi$ and $\psi$ are conjunctions of a known formula and zero or more unknown predicates. The goal is to construct a liquid assignment that satisfies all of the predicates, or determine it is unsatisfiable.

Synthesis rules are constructed from the typing judgements. When synthesis is attempted, the rules for generating fix-point definitions and abstractions are used. If the given goal type is scalar then the system begins by enumerating all well typed elimination terms, and attempting to solve constraints along the way. If the constraints are trivially true then a solution has been found, if they are inconsistent the term is discarded, otherwise a conditional is generated and synthesis of the false branch is attempted. Once all well typed expressions be enumerated the system attempts to synthesise a pattern matching definition using an arbitrary elimination term.

The suite of benchmarks used to evaluate Synquid is considerably larger than previous systems, with 64 definitions. Synquid was able to synthesise every test attempted. Those which had been attempted by previous systems were synthesised considerably faster by Synquid. The results show that the extension of the type system with extra information not only allows specifications to be stated more precisely, but to significantly improve performance.

## 3.6 Dependent Type Driven Program Synthesis

The Idris programming language has proof search functionality built in, with the recent release of Idris2 this has been improved. The internal representation of the language is similar to that of the TinyIdris system, however the full Idris 2 implementation has much more information available, much of this is due to the more sophisticated type system, along with file information. The algorithm follows certain steps. When given a hole, attempt the use of local variables, this step has been refined by projecting the elements of pairs. If that fails then the term is matched on, if the term being synthesised is a pi binder, then synthesis is then we attempt to synthesise the return type and if successful return a lambda for the type of the term inside the pi. If successful. If the term is a type constructor then for every data constructor, attempt to construct an application of that constructor and attempt unification, if this succeeds, attempt to solve the remaining holes. If all of the above fails, attempt synthesis using a recursive call with a structurally decreasing argument.

The system also includes heuristics, such as checking the number of arguments used from the left hand side, to determine the 'best' term, amongst others, which have not been formally detailed.

The implementation has not been formally tested in the same way as the other systems presented. Two major differences between this system and the previous three presented is the lack of a full enumeration of the context. While this may increase the number of terms synthesisable, this system is also implemented as part of a full programming language as opposed to a standalone tool, this may introduce performance issues to the synthesis that may not hinder the previous tools.

# 4 The Synthesizer

At its heart, program synthesis is a search problem. The search space consists of every possible way to construct a term from the given context and environment. Following a naive approach will quickly become infeasible, thus we must find ways to restrict the search space to one that can be enumerated within a reasonable amount of time. Since there are many more incorrect programs than there are correct programs, using the type checker to do this seems a good place to start.

This approach consists of using a set of synthesis rules, that generate constraints and subgoals, propagating the constraints down through the subgoals. Once no more rules apply, the enumeration should begin, using typing information and constraints to restrict the possibilities as much as possible.

If this results in a valid term being constructed, the term should be propagated back up, and combined to construct the main goal term. If all possible terms have been enumerated and none are valid then the terms on the left hand sides can be matched on to provide the type checker with more information.

### 4.0.1 Constraints

Constraints exist in several forms. In the TinyIdris system, if the type provided is one or more applications `App`, the type to be synthesised is the function type, with the added subtyping constraints that the final type unify with the application. Constraints are also generated during unification, these constraints result in the construction of a branching term that handles the case where the constraint is satisfied and the case where it is not.

Sub-typing constraints are propagated down and help to reduce the search space. Where branching constraints are generated we require that all possible outcomes must result in a valid term, and if more constraints are generated then they are valid, in order for a top level branching term to be synthesised.

### 4.0.2 Termination checking

When synthesising the definition for a function, the function will already be stored within the context, since it will return the type we are looking for it would be enumerated as a possible definition, this creates obvious termination problems. For this reason, when making a recursive call, there must be some check to ensure at least one of the arguments provided is structurally decreasing, and will eventually stop. Within the TinyIdris system this will require the implementation of a termination checker, as this has not been implemented in the language. The synthesis algorithm will also need to know the function that is currently being defined, which is also not available to it within the current implementation of the system.

The alternate form that termination checking takes is checking that co-inductive definitions, although potentially non terminating, are required to be productive. For simplicity this is omitted here and left as a potential future development.

### 4.0.3 Heuristics

It is possible for the algorithm to construct terms which type check, however are still incorrect, especially when there is a lack of type information. Synthesis algorithms can be optimised with heuristics to help prevent this. One of the issues can be seen in the map example with lists, the synthesis algorithm does not have enough information to know that the `Nil` constructor is incorrect, however a check can be added to select the most likely term. Returning the term which uses the most of the arguments from the left hand side, this is based on the idea that if an argument is provided then it is probably intended to be used.

Another potential issue is repeated application of functions, for example, when synthesising `Nat`, if we have a `n:Nat n` would be valid, as would `suc n`, and `suc (suc n)` and so on. One potential way to improve this is to track the number of times a function is called at any given stage, iterative deepening will address this from a termination standpoint.

Since pattern matching may lead to redundant matches, ensuring that this only happens after all available terms have been enumerated.

Another common optimisation used has been to ensure terms to be synthesised are in $\beta$-normal-$\eta$-long form. This reduces the number of redundant terms for which synthesis is attempted. If multiple reduction rules apply to a term, then there exists a term that can be reached from each reduction through the application of more reduction rules. cite:10.5555/2788232 By operating only on normalised terms we avoid this branching.

Finally, stopping enumeration once a valid term has been found will cut down on the number of enumerations, however this may lead to worse results if other heuristics are determining the 'best' possible term rather than simply taking the first found.

## 5 Testing and Evaluation

A test suite will be created consisting of examples that should each test a specific area of the synthesis algorithm. This should be in the form of several test scripts containing holes, along with a solutions files containing the completed definitions. For each test, the synthesised term should be recorded, along with the number of steps taken to reach the solution and whether or not the solution was indeed the intended one. After each refinement of the algorithm these tests should be re-run and compared. Different depth sizes should also be tested, with the attempt to find a balance between time taken and number of terms synthesised.

The test files are based off of benchmarks from earlier works containing some of the simpler and more difficult, based on previous performance. The tests will also contain some benchmarks not seen previously to compare the performance of the more general searching method against the tactic based approach implemented in Agda.

- Vectors
- Lists
- Equality
- Sorting algorithms
- Self balancing trees

The more basic examples are lists and trees, these will be used to test the use of recursive and higher order functions, these will also test any pattern matching capabilities implemented. The Equality tests have the purpose of comparing the algorithm to the tactic based approach seen in Agsy. The sorting and self balancing trees will be some of the more challenging examples that will test how optimally the algorithm performs as there should be a much more noticeable gap if there are performance issues.

The questions being asked when evaluating the system will include:

- How does the system compare to existing languages that support program synthesis?
- How does the system compare to existing systems that are designed specifically for synthesis?
- Is the synthesis functionality fast enough to be usable as part of a workflow?

bibliography:ProgressReport.bib