TinyIdris Program Synthesis

CS408 Progress report

Scott Mora - 201816428

March 15, 2021

1 Introduction

Programming often involves implementing low level details that can become 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 system to guide the synthesis algorithm towards a solution by reducing the number of terms enumerated. This approach targets a common pattern seen in functional languages, recursion. Data types can often follow similar inductive patterns, similar to that of Lists. Dependently typed programming languages, though not confined to being functional languages, are typically structured in this functional way, and allow a greater level of usable information to be encoded within the types of programs.

This style of programming has several benefits. By encoding more information within types it allows the type checker to pick up on potential bugs that may not be caught until runtime otherwise, it also allows certain impossible states to be un-representable within programms, ensuring the correctness of code and limiting the use of fatal exceptions. Typically functional programs encourage more code reuse, and functional definitions can be considerably shorter than their imperitive counterparts, which can result in shorter, more readable code.

To see how this can impact program synthesis, we look at an example, and how increased type information increases our chances of success. We define the `List` data type, and aim to synthesise a definition for the function 'map' which. When given a function as a parameter, and a list, map applies the function to each element of the list, a commonly used pattern in functional programming. The list argument has been pattern matched on, and the holes to be synthesised have been added.

```
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
```

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). This paper will outline an approach to extend TinyIdris with program synthesis functionality via commands in the repl.

2 The TinyIdris System

TinyIdris is a dependent typed programming language with limited support for implicit arguments, the language is a scaled down version of the Idris2 programming language, developed for teaching the implementation of programming languages. The language is not compiled, instead, it comes in the form of an executable program, which accepts the name of a Tiny Idris '.tidr' source file to be processed by the system. The source code is converted into the internal representation of the language, type checked, and each definition is stored in memory. After the file has been processed, a read eval print loop, or repl is presented, which accepts expressions from the language, evaluates them, and prints out the result.

```
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)))
>
```

Listing 1: Evaluating 2 + 1

2.1 The Source code

Tiny Idris source code files consist of three top level global declarations, Data Types, Type Signatures, and Function Definitions, each of which has their own scope.

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

add : Nat -> Nat -> Nat
add Zero m = m
add (Suc n) m = Suc (add n m)

multiply : Nat -> Nat -> Nat
```

2.1.1 Type Signatures

Type signatures outline the shape of the data that is being transformed. Written `f:a_1 ... a_n -> r`, where `f` is the name, `a_1 ... a_n` are the parameters, and `r` is the return type, read "f has type, a_1 to ... a_n to r=. Types in dependently typed languages are first class, meaning it may take in any term, not just types, this includes function types, and can be taken further, however this is outwith the scope of this paper. 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_1 ... a_n -> (r, r', r'')`. Type signatures are present in both of the following constructs.

2.1.2 Data Types

Data type declarations define a type, and describe how to construct values of it. They consist of a type signature for the type being defined, known as a type declaration, and a set of data declarations, which are type signatures for each of the constructors. The data type declaration begins with the `data` and is followed by the type declaration. The data constructors are defined in the where clause that follows the type declaration. A type may have zero or more data constructors.

This style encourages inductive definitions, where the base case is defined, from which the other cases are built up, as seen in the `List`, `Vec`, and `Nat` examples. Defining Booleans however, we see that this need not always be the case.

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

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

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 are split into cases, each of which contain a left hand side and a right hand side. The left hand side is an application of the function to the arguments to which it is being applied, 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 of arguments can be matched on, however it is enforced that every possible case for each argument is covered. Not all arguments that a function takes in must be listed on the left hand side, if certain perameters are left out, the return type will be of the form `p_1 ... p_n => r` where n is the number of remaining arguments, in this case a lambda expression can be constructed, taking in the remaining parmeters.

```
add : Nat -> Nat -> Nat
add Zero = \ m => m
add (Suc n) = \ m => Suc (add n m)
```

2.1.4 Parametricity

We define lists inductively, in the same fashion as natural numbers, by first building up from the base case, and successivly adding an element.

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

Similarly to that of data constructors and functions, types may also have parameters. 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, supporting code reuse.

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

In dependently typed languages, 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 (Suc 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.

Using dependent types allows for complex properties of data to be expressed, and checked by the type checker, these include writing proofs, creating 'views' that look at data in specific ways, and embedding specific properties of data within the type, as shall be seen in the next section, where it is enforced that terms are well scoped by construction, within the TinyIdris system.

```
mapAppendDistributive : (f : a -> b) -> (x : List a) -> (y : List a) ->
            map f (x ++ y) = map f x ++ map f y

data Divides : Integer -> (d : Integer) -> Type where
     DivByZero : Divides x 0
     DivBy : (prf : rem >= 0 && rem < d = True) ->
         Divides ((d * div) + rem) d
```

2.1.5 Differences from the Examples

For simplicity of the examples, the examples displayed have not been valid 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)
```

There are two main differenced occuring from TinyIdris's lack of full support for implicit arguments. 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, requiring each name found in the application to be first brough into scope with a pattern variable. More complete languages also use other features such as let bindings, case splitting and the with idiom, all greatly increasing the expressiveness of the language.

2.2 The Raw Implementation

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

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

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

   data ImpData : Type where
      MkImpData : (n : Name) ->
            (tycon : RawImp) ->
            (datacons : List ImpTy) ->
            ImpData

   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
```

Expressions from the language are represented as by the RawImp data type. Each name referenced within a term will be done so via an IVar term, eg `x, xs append`. `Type`, is the type of types, and Pi types are the type of terms that take argumemts, and can be read as saying "forall elements of the argument type, the return type holds". `Type`, the type of types, also has type `Type` in the language, this can present isssues via Girard's paradox, however this will not affect the results, as it can be difficult to accidentally come across issues.

IPatVars are found in each clause of a pattern matching definition taking in a name, term and the scope, these are the =pat x : y, ... => = lines in syntax, the scope type is either another IPatvar or an IApp, of the function being defined to it's arguments. IApp represents an application of a function to an argument, and are found of the left hand side of clauses, and the right hand side where a function application is used.

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 declarations, each declaration is then processed by the elabourator. Elaboration is the process of converting RawImp terms to terms in the core language, and will be discussed in more detail later.

Internally, declarations are stored in the context is a map of names to global definitions, where each globaldef has a type, in the form of a closed term, which is a term with no names in scope, and a definition, the `Def` datatype.

Along with each of the definitions discussed earlier, they can also be None, which is the definition of a type signature without an accompanying function declaration, a hole, or a guess, which are used during unification. Which will be discussed later. Constructors are stored with a tag to differentiate them, and an arity for convenience, the definition of each is straightforward.

Expressions in RawImp are converted to the `Term` datatype, which is indexed by a list of names that are currently in scope, to ensure that all terms in the internal representaiton are well scoped.[1]

```
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
   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
   TType : Term vars
   Erased : Term vars
```

IVars will be converted to one of two possibilities. 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 describe 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 terms which have been erased.

Local contexts, are stored in an Environment, represented by the `Env` data type.

```
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, such as normal forms, or closures, 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 referenced, each time a binder is added to the environment, then a name that may be referenced is added to the environment with it.

2.4 Values

Values within the TinyIdris system are in Weak 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.

[1] The `0` found in the `IsVar` argument is a quantity, and can safely be ignored for our purposes. For more information, see cite:BibEntry2020Nov.

```
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

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
```

Values being in Weak Head Normal form means that the outermost part has been evaluated, however the arguments may remain unevaluated. The outermost part, or head, is something that may be applied to arguments, such as a constructor, or function, referenced, the head consists of a name referencing it, or a metavariable created during unification.

The arguments are stored as 'thunks' within the `Closure` data type. They contain a term, and an enviroment that the term should be evaluated in, so that it can be evaluated when the system is ready to.

Values can also be binders, which cannot be evaluated until the argument being taken in is known.

2.5 Process

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

```
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 has two main purposes, the first is to convert RawImp terms to Terms in the core language, stripping away features from the high level implementation that are not present at the lower level, such as implicit arguments. In the full implementation there are sevral more features stripped away at this stage. The second purpose is to perform type checking.

   Elaboration is implemented via two functions, checkTerm and checkExp (for checkExpected). Checking a term, when provided with an environment, RawImp term, and possibly an expected type, will return the checked Term in the core language with its type, if type checking fails then an exception is thrown.

Glued variables are simply terms, paired with their normal form for convenience. The checkTerm function, elaboration proceeds by breaking down terms into their individual componenents, checking each component which provides the appropriate `Term` and putting them back together as a `Term`. checkExp is then called with the expected value and the generated term. For example, when checking the term `(IApp f a)`, the function is checked, if it's term is a `Bind` with a pi binder, the argument type is checked, and checkExp 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 the argument is checked, the environment is extended with the resulting 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

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

Unification has the purpose of whether two terms are substitutable. It operates by recieving an environment, and two terms scoped with the environment, along with the global context and the unification state, `UState`. The unification state maintains information such as the holes that are present in the program, along with guesses made by the unification algorithm, and constraints on the equality of terms.

The Main purpose of unification is to check if two terms could be convertable. Rather than simply checking if two terms are equal, unification attempts to generate a set of constraints that would lead to the two terms being equal, if the constraints are unsatisfiable then unification fails, otherwise the constraints are added to the unification state.

Unification proceeds by reducing the terms being checked to values, and checking the constructors used for each term, in the event of two binders, if the terms being taken in unify, then a name is generated and a binder talking in one of the terms is added to the environment, in which unification is attempted with the scopes, if both terms are a constructor then it is checked that the constructor names are equal, and then unification is attempted with the arguments.

Otherwise unification succeeds if the two terms are convertable, which checks of they are equal. If multiple terms are unified, the constraitnts generated are unioned, and returned.

If one or more of the terms is an application containing a metavariable then it attempts to solve and instantiate the meta, this may solve other holes, or generate new ones, which are added to the unification state. The algorithm returns a `UnifyResult`, which consists of a boolean specifying if any holes were solved in the process. At various points during type checking, including where holes have been solved, then solving the remaining constraints is once again attempted.

3. Processing

Processing follows a similar pattern for each type of declaration.

The `ImpTy` 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.

To process `ImpData` Data Types, first the type constructor is checked in the empty environment, and a new definition is added to the context, with the `Def` as a TCon, for each data constructor, it is similarly checked in the empty environment and for each a new DCon definition is added with

their type. Each constructor has a tag, between 0 and the number of constructors, distinguishing them.

The most work is done while processing definitions, each clause is split into a left hand side of the form `pat a : A, b : B => f a b`, and a right hand side constructing a term of the return type. Processing occurs by first checking the term of the left hand side, it then moves through each pattern in the term and type generated for the left hand side and adds them to the environment, which is used to check the right hand side, using the remaining type as the expected type of the rhs. A clause is then made using the environment, left hand side term, and right hand side term.

Once each clause has been processed, the algorithm then generates a case tree for the given clauses, which is then stored within the context as a PMDef, since the type signature must have been processed previously, the name and type will be stored in the context with the definition None, so the existing definition is updated.

3 Related work

There has been several creative attempts at synthesising programs from many fields within computer science, such as the machine learning, programming languages, and program verification communities. Some of the research presented here has since been improved with the introduction of quantitative (resource) types, where values are annotated with a multiplicity, stating how many times it may be used, this has been shown 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. The language supports many of the same features as Idris, such as hole driven development with interactive typing information, and many other constructs common to dependently typed programming languages. Agsy is a tool developed and currently implemented as part of the Agda interactive development system. The language features holes of the form `{ } 0`, where the following number is unique to that hole, with the cursor inside the hole the user can invoke the tool by pressing "C-a", alternatively, it exists as a stand alone tool for testing. 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 does mean it is limited by the expressiveness of the tactic language. However this may be considered a benefit, as more general approaches may not be as effective at synthesising solutions that require specific knowledge of the problem domain, and could lead to the tool being extended in similar ways to that of Coq's tactics language.

3.2 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 operates on a subset of Scala, and 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.

```
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 resulting in the desired 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.

3.3 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.

```
let list_stutter : list -> list |>
  { [] => []
  | [0] => [0;0]
  | [1;0] => [1;1;0;0]
  } = ?

let stutter : list -> list =
let rec f1 (l1:list) : list =
  match l1 with
  | Nil -> l1
  | Cons(n1,l2) -> Cons(n1, Cons(n1, f1 l2))
in f1
```

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 β-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.

3.4 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 β-normal-η-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 ϕ and ψ 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.

```
data RList a <r :: a -> a -> Bool> where
  Nil :: RList a <r>
  Cons :: x: a -> xs: RList {a | r x _v} <r> -> RList a <r>

termination measure len :: RList a -> {Int | _v >= 0} where
  Nil -> 0
  Cons x xs -> 1 + len xs

measure elems :: RList a -> Set a where
  Nil -> []
  Cons x xs -> [x] + elems xs

type List a = RList a <{True}>
type IncList a = RList a <{_0 <= _1}>

leq :: x: a -> y: a -> {Bool | _v == (x <= y)}
neq :: x: a -> y: a -> {Bool | _v == (x != y)}

sort :: xs: List a -> {IncList a | elems _v == elems xs && len _v == len xs}
sort = ??

sort = \xs .
  let f0 = \x2 . \x3 . \x4 .
  match x4 with
    Nil -> Cons x3 Nil
    Cons x12 x13 ->
    if x3 <= x12
      then Cons x3 (Cons x12 x13)
      else Cons x12 (f0 x13 x3 x13) in
  foldr f0 Nil xs
```

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 succincly, but to significantly improve performance. The system still suffers from downsides, specifications can still prove to be bulky, which for simpler functions can be longer and more difficult to produce than the function definition itself. The tool is also limited since it is not related to an existing language, and thus cannot be used in any practical way. The tool has since been replaced by ReSyn, which extends the language with 'Liquid Resource Types'. This has been shown to increase the effectiveness of the tool, however suffers from the same issues as Synquid.

3.5 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, after traversing the binders, 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 Synthesis Tool

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.

The tool has two main modes of operation. Synthesising individual terms, or full pattern matching definitions. The TinyIdris repl has been extended with an command, `auto`, which takes a name. If the name is in the context with no definition then the tool attempts to synthesise a pattern matching definition. To support synthesis of individual terms, the language has been extended with 'holes' of the form `?hole_name`. The holes name can be provided to the tool, which will attempt to synthesise a single term. The tool returns a string of TinyIdris syntax, which can be inserted into the source file. If synthesis fails then an error is returned.

4.1 Extensions to the language

The language has been extended with user inserted holes of the form `?hole_name`. A constructor `IHole : Name -> RawImp` has been added to the RawImp data type and the parser extended to accept the new syntax. The unification state has also been extended with a sortedmap of Names to user holes, when IHoles are encountered during processing they are added to the map. The `Def` type was extended with a `MetaVar : (vars : List Name) -> Env Term vars -> (retTy : Term vars) -> Def`, the definition stores the local environment that the metavariable is definied in along with it's return type, and the names in scope. User provided holes should only appear on the right hand side of pattern matching definitions, thus there should always exist an expected type whenever an `IHole` is elaborated, during elaboration of holes, we get the expected term, and use it to generate a new `Meta` term, with a `MetaVar` definition using the environment and expected term.

To improve performance, certain other information has been added to the unification state, a sorted map of names and This reduced the number of terms enumerated by the synthesiser as the context contains metavariables created during unification, causing the context to quickly blow up in size with values which could not lead to a candidate term. The processing of definitions has been extended to add new types and functions to these maps. Functions are added after the type is processed, allowing them to be used during synthesis, without having been implemented.

The unification of TinyIdris fails with an error when a term is ill typed, in regular use this is the desired outcome, however when synthesising terms, the unification being performed is unsafe, since it list almost guarentted to be used on terms which will not unify, thus it has been extended to handle failure. The uninfication algorithm was also extended to support the unification of binders, which is nesccesary when synthesising binders.

Synthesis also create the need for new error messages, as certain exceptions may occur that did not before, such as the name being provided not being found in the context, or having an invalid definition.

4.2 Synthesising Individual Terms

A `Search` is represented internally as:

- A `Nat` depth
- The name of the expression being synthesised.
- The `Env` Local enviroment.
- The `RawImp` left hand side of the target term.
- A `Term vars` target type.

The depth is introduced to avoid termination issues. When synthesising a term of type `Nat`, the depth first nature of the search would lead to termination issues without a specified depth, for example when synthesisng a `Nat`, it would attempt the `Suc` constructor, which takes a `Nat` argument, and so on.

An initial depth of 4 has been found to be sufficiently deep to provide useful results, and not hinder perfomance.

The name, enviroment and target type play an active role in synthesis, while the left hand side is used to check the term synthesised is structurally different from that on the left hand side.

```
record Search (vars : List Name) where
 constructor MkSearch
 depth : Nat
 name : Name
 env : Env Term vars
 lhs : RawImp
 target : Term vars

synthesise : {vars : _} ->
       {auto c : Ref Ctxt Defs} ->
       {auto u : Ref UST UState} ->
       Search vars -> Core (List (Term vars))
```

Not all terms may be synthesised, we are only able to construct terms with a type as a type. The algorithm begins by inspecting the target type. If the type is a pi binder then we may construct a lambda with the argument type, and attempting to synthesise the scope.

If the type is of type `Type` then we are able to synthesise valid terms by using anything of type Type in the context, however, this will lead to sevral incorrect answers being generated. Since types are passed in explicitly as patterns, we restrict the usable types to only those that have been passed in, since generally these will be the desired ones, as they frequently occur while synthesising arguments to candidate terms, since every argument must be explicitly passed.

After moving through each pi binder, the resulting scope must be an application of a type constructor to zero or more arguments. If this is not the case, or the maximum depth has been reached, then the algorithm will check the local variables in scope for a term of the given type, which will only require a maximum of two passes of the environment, before terminating.

If the term is a valid application or type constructor then synthesis is attempted first by checking the local variables and then trying type constructors and functions. When synthesising a term, a list of potential candidates will be returned, which may then be ordered based on some heuristic.

4.2.1 Searching Locals

The algorithm first attempts to check the local environment for valid terms, this is mostuly common when defining the base case of a recursive function, or when synthesising arguments within attempting definitions. The process is split into two stages.

Since only PVar and Lam binders result in a usable term being brought into scope the first stage consists of traversing the environment and filtering out all of the un-usable binders, if a term is valid then we must construct a `Local` term referencing the name, the list of usable local variables is then passed to the second stage, which traverses the list and gets the binder from the environment. Unification is attempted between the target and the type of the binder, if no constraints are generated then the Local term is accepted, and the rest of the environment is checked.

If the binder unification fails, and the depth is non-zero, then a the type of the binder is checked, if it is a function type then the arguments are filled by metavariables and unification is once again attempted, if this succeeds then terms for each of the functions arguments are searched for, if successful an application is returned. This is nesccesary for higher order functions to be synthesised.

4.2.2 Searching Globals

Synthesis via data constructors and function definitions both follow the same process. Data constructors are attempted first, under the assumption that this will be the more likely solution. However this may be overridden by any ordering heuristic used.

```
tryDef : {vars : _} ->
     {auto c : Ref Ctxt Defs} ->
     {auto u : Ref UST UState} ->
     Search vars -> Name -> NameType ->
     Term [] -> Core (List (Term vars))

tryIfSuccessful : {vars :_} ->
          {auto c : Ref Ctxt Defs} ->
          {auto u : Ref UST UState} ->
          (Search vars) ->
          Name -> NameType ->
          NF vars -> Core (List (Term vars))
```

When attempting to use a global definition, the problem is represented as a name, nametype and closed term for the type of the definition. Each type will be of the form `p -> ... -> p_n -> r` where $p_1 \ldots p_n$ are arguments to the function and r is an application of a type constructor to zero or more arguments. To avoid synthesising definitions which will result in an invalid term, metavariables are constructed for each binder and unification is attempted between the resulting type and the target type. If successful, a depth first traversal of the arguments takes place, synthesising terms for each, otherwise the search is stopped, and the algorithm moves on to the next definition to be attempted.

When synthesising arguments, it is possible that arguments deeper into the binder may depend on arguments that have been previously passed in, therefore branching occurs as we move down through the binder. For each branch, the scope of the binder is normalised using the synthesised term to construct a closure and unification is attempted between the scope and the target in the same fashion outlined above, if this fails then the branch is killed, otherwise the process is repeated for the scope. If the term passed in is not a Pi binder then we have reached the end of the arguments, and a Reference to the type constructor is returned, to which the synthesised terms may be applied.

4.2.3 Structural Recursion Checking

The structural recursion check is conservative, in that it does not reduce terms, so may deny terms which are in fact structurally different. It checks that each name present in the right hand side term is present in the left hand side term, if the right hand side is a binder then it is assumed not to be different. The check has been designed this way to avoid terms being allowed because their arguments are different, however they are simply in a different order.

4.2.4 Final Ordering

4.3 Synthesising Definitions

The problem of synthesising definitions utilises the synthesiser for terms, however has the added complexity of introducing pattern matching, and recombining the resulting terms.

The only information avaliable initially is the type signature of the function, from that, an initial left hand side RawImp term that has no case splits is constructed. As a heuristic no synthesis is attempted at this stage, as typically functions are longer than a single case, and this could lead to valid, however incorrect definitions being synthesised, any term which may be correctly synthesised at this stage will also be correct after splitting the cases, thus the negatives of this decision are outweighed by the positives.

4.3.1 Pattern Matching

The splitting algorithm recieves either a singleton list containing the initial left hand side, or a list of multiple which have been generated by a previous split, each of which is split again. Each lhs at the beginning will have at least one unique term, which will not be split on, so every generated left hand side will be unique. Any invalid terms generated are filtered out by the type checker.

The terms split on will be the leftmost type constructor. Following a strict left to right split will lead to multiple redundant splits, this problem is exaserbated since each argument must be passed explicitly. To reduce this, a traversal of the generated split occurs, which uses the type infomation available from the initial split to fill in any implicit arguments generated, and if the new information provided by the split results in any deeper terms being forced into a certain pattern. Again the resulting left hand sides are checked by the type checker to ensure their correctness, the traversal repeats for each new split

enerated. Each lhs is split into its own world, for which synthesis is attempted. If each world results in a valid term, then combining the worlds into a definition is attemted, if this results in a valid definition, it is accepted, otherwise each world is further split, a potential future optimisaiton could be to only split the worlds that are unsuccessful.

For each world to be synthesised the RawImp left hand side must be converted into an environment, and a term, a list of candidates are then synthesised for the target term, synthesis is attempted for every other world, if any fail then the process is killed, otherwise the top term synthesised is taken to avoid a blow up in defs being synthesised. The selected term is converted into a clause, which is returned with each of the other clauses synthesised.

4.3.2 Combining Clauses

TinyIdris has a built in CaseTree representation for clauses, when the synthesiser returns a set of clauses, it is combined using this, ensuring the correctness of the definition. Each definition is then returned, along with the list of clauses to be converted into source code for the user.

After a valid term has been synthesised, be it a list of clauses or a valid term or full definition, this is converted back into a string in TinyIdris syntax for the user to insert into the source file.

```
Running Auto Search:
append : (a : Type) -> (n : Nat) -> (m : Nat) -> (xs : (Vec a n)) -> (ys : (Vec a m)) -> (Vec a (add m n))
pat a : Type, x01 : Nat, m : Nat, ic10 : a, ic11 : Vec x01 a, ys : Vec m a =>
   append a (S x01) m (Cons a x01 ic10 ic11) ys = Cons a (add m x01) ic10 (append a x01 m ic11 ys)
pat a : Type, m : Nat, ys : Vec m a =>
   append a Z m (Nil a) ys = ys

> auto v01
Running Auto Search:
Nil b
> auto v02
Running Auto Search:
No match
> 
```

Listing 2: Synthesising a definition, term, and failing on a term.

5 Testing

Testing consists of running the tool on functions, it will be evaluated based on the resulting functions correctness, along with more subjective notes. The results are compared to that of the current tool within the Idris system, along with Agsy, Agda's auto tool.

The test suite has been split into multiple categories, each testing a specific purpose. Each individual test has been selected from the test suite used to evaluate Agsy, and the test suite used to evaluate Synquid, a synthesis based language. Each has been converted into TinyIdris, Idris, and Agda. Since Synquid requires extra infomation for synthesis, the results are not direcly comparable to the system.

- Lists
- Vectors
- Equalities
- Sorting

The list tests will be used to evaluate effectiveness of the tools ability to synthesise terms without relying on detailed type information.

Vectors will contain the same examples as lists, with the added type information of the length built in, this will be the middle ground, where some information is available, however it is not overly restrictive.

Sorting uses sorted vectors to test the effectiveness of the tools when there is more complex type information avaiable.

Testing equalities focuses on the effectiveness of pattern matching refining the left hand side to reach a valid term.

Testing individual terms can be carried out individually using the command `t` with the name of the hole, or in batches using the `test` command, for which a predefined answer file must have been created. Testing of full pattern matching definition is completed on an individual basis.

```
> test
Running tests:
Test: v01 | Result Success | Expected: ys | Actual: ys
Test: v02 | Result Fail | Expected: Cons a (add n m) x (append a n m xs ys) | Actual: Cons a (add m n) x (append a n m xs ys)
Test: v03 | Result Success | Expected: Nil b | Actual: Nil b
Test: v04 | Result Fail | Expected: Cons b n (f x) (map a b n f xs) | Actual: No match
Test: v05 | Result Success | Expected: Nil a | Actual: Nil a
Test: v06 | Result Success | Expected: Cons a n x (replicate a x n) | Actual: Cons a n x (replicate a x n)
Test: v07 | Result Success | Expected: ys | Actual: ys
Test: v08 | Result Fail | Expected: drop a n m xs | Actual: append a Z m (Nil a) (drop a n m xs)
Total successes 5
> t v03
Running One Test:
Test: v03 | Result Success | Expected: Nil b | Actual: Nil b
> 
```

Listing 3: Evaluating 2 + 1

6 Evaluation

6.1 Initial Perfomance

We now present the results of each test, for each test, with each of the four systems being compared. Attempts were made at synthesisng without case splitting initially, both Idris and Agda failed on each test, the testes were repeated after the case splits were done manually, in the order most likely to present a result.

6.1.1 Lists

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

6.1.2 Vectors

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

6.1.3 Sorting

Problem	TinyIdris	Idris	Idris2	Agda
list to vector	Fail	Pass	Pass	Pass
vector to list	Fail	Fail	Pass	Fail
insert				
sort				

6.1.4 Equalities

Problem	TinyIdris	Idris	Agda
plus commutes	Fail		
plus Suc	Fail		
symmetry	Fail		
transitivity	Fail		
congruence	Fail		
list(vec(list)) = list	Fail		
vec(list(list)) = vec	Fail		
disjoint union apply	Fail		
a = not not a	Fail		
not not not a = not a	Fail		