

TinyIdris Program Synthesis

Marking Scheme: Experimentation

Scott Mora - 201816428

Submitted for the Degree of B.Sc. (Hons) in Computer Science.

Except where explicitly stated all the work in this report, including appendices, is my own and was carried out during my final year. It has not been submitted for assessment in any other context.

Contents

1	Introduction	4
2	The TinyIdris System	6
2.1	The Source code	7
2.1.1	Type Signatures	7
2.1.2	Data Types	7
2.1.3	Function Definitions	8
2.1.4	Parametric Definitions	8
2.1.5	Differences from the Examples	9
2.2	The Raw Implementation	10
2.3	The Core Representation	11
2.4	Values	12
2.5	Process	13
2.6	Extensions to the language	15
3	Related work	16
3.1	Automated Theorem Proving in Agda	16
3.2	Synthesis Modulo Recursive Functions	17
3.3	Type and Example Directed Program Synthesis	18
3.4	Program Synthesis from Polymorphic Refinement Types	20
3.5	Dependent Type Driven Program Synthesis	21
4	The Synthesis Tool	23
4.1	Synthesising Individual Terms	24
4.1.1	Searching Locals	25
4.1.2	Searching Globals	25
4.1.3	Structural Recursion Checking	27
4.2	Synthesising Definitions	27
4.2.1	Pattern Matching	27
4.2.2	Combining Clauses	28
5	Detailed Implementation	29
5.1	Synthesis	29
5.1.1	Individual Terms	29
5.1.2	Pattern Matching Definitions	31
5.1.3	Pattern matching	33
5.1.4	Un-Elaborating and Re-Sugaring	36
6	Evaluation	37
6.1	Test Suite	37
6.2	Testing Functionality	37
6.3	Testing Plan	38
6.4	Performance	38
6.5	Evaluation of performance	39
6.6	Comparison to other systems	40
6.7	Changing Heuristic	40
6.8	Conclusions	41
7	Appendix A - User Guide and Maintenance	42

I am extremely grateful to Fredrik Nordvall Forsberg, of the University of Strathclyde for his guidance, patience, and support throughout this project.

I am also grateful to Edwin Brady of St. Andrews University for creating many useful resources making this project possible.

Many thanks to Fredrik Lindblad, Marcin Benke, Etienne Kneuss, Viktor Kuncak, Ivan Kuraj, Philippe Suter, Armando Solar-Lezama and Nadia Polikarpova, who inspired this project through their ideas, and previous work.

1 Introduction

Programming often involves implementing low level details that can become repetitive and tedious, such as traversing data structures within recursive functions, and implementing multiple data types of a similar structure. 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 tool towards a solution by reducing the number of terms enumerated. This approach targets a common pattern seen in functional languages, recursive pattern matching definitions. Dependently typed programming languages, though not confined to being functional languages, are typically structured in this functional way, and allow a greater amount of information to be encoded within the types of programs.

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

To see how this can impact program synthesis, we look at an example, and why type information increases our chances of success.

We define the generic `List` data type, using the `data` keyword, and define two data constructors, `[]` (read ‘Nil’) and `::` (read ‘Cons’) explaining how to construct elements of the list type. The Nil constructor takes no arguments; the Cons constructor takes an element, and a list, which represents the list containing the appended element. The list may contain elements of any type, the type of which is passed in implicitly, represented by the argument `a`.

Functions describe how to transform a set of inputs into an output, and consist of a type signature and a pattern matching definition. The type signature describes the type of each argument, and the data type being constructed as the output. Pattern matching is used to inspect which data constructor has been used to construct a certain type, where the function name is followed by the list of arguments to which it is applied in the given case, arguments may or may not be split, however if they are then each possible data constructor must be considered. Given the type signature and patterns to be inspected, we attempt to synthesise a definition for the ‘map’. When given a function and a list as it’s arguments, 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 representing the terms to be synthesised have been added, denoted by `?name`.

```
data List : Type -> Type where
  [] : 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 Cons constructor. However, inserting the term Nil 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 possible incorrect answers, reducing the number of incorrect answers returned, and increasing the speed of the algorithm 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. Vectors are lists, indexed by their length, meaning that, for some type `a`, `Vector 5 a` is a different type to `Vector 2 a`, this is enforced by construction, as the empty vector is indexed by the number 0, and with each element added, 1 is added to the index.

```

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

```

It is now enforced within the type signature of `map` that the length of each of the output list must equal that of the input list. The process for the first hole remains unchanged, since there is only one term satisfying `Vector 0 b`, `Nil`. In the `Cons` case however, the empty vector is no longer a valid term, since the type `Vector 0 b` \neq `Vector (1 + n) b`, allowing us to find the correct term.

TinyIdris is a small dependently typed programming language. Users interact with the TinyIdris system by loading their source files into the TinyIdris repl (Read Eval Print Loop), allowing them to enter expressions to be evaluated. This project will outline an approach to extend TinyIdris with program synthesis functionality via commands in the repl.

2 The TinyIdris System

TinyIdris is a dependently typed programming language with limited support for implicit arguments. The language is a scaled down version of the Idris2 programming language, developed by Edwin Brady with the purpose of teaching the implementation of programming languages [3]. The language is not compiled, instead, it comes in the form of an evaluator, accepting the name of a TinyIdris '.tidr' source file to be processed by the system, and presenting a Read Eval Print Loop, which accepts expressions for evaluation, as displayed in figure 1.

```
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)))
> █
```

Figure 1: Evaluating $2 + 1$

Figure 2 displays the processing order of the system, where the system parses the source code into the internal high level representation of the language, RawImp. High level features such as implicit arguments present in RawImp are stripped away and the terms are type checked, resulting in the core language representation, TT. The process of type checking consists of evaluating terms, and ensuring their types are as expected, at which point their representation may be returned. When processing is complete, the resulting terms are stored in memory as definitions, which may be used later when evaluating expressions, in figure 1, this is the case for the Nat type, and add function.

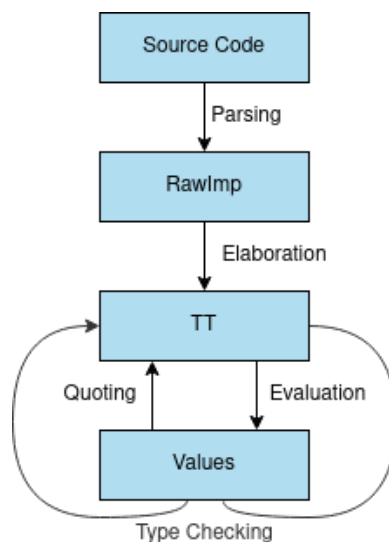


Figure 2: TinyIdris processing order

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 local 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 $f : a_1 \dots a_n \rightarrow r$, where f is the name of the construct being defined, $a_1 \dots a_n$ are the parameters, or arguments, which describe the types of each input to the construct, and r is the type of the functions output, read “ f has type, a_1 to $\dots a_n$ to r .” Types in dependently typed languages are first class, meaning parameters may be any term, not simply data types, this includes function types, data constructors, case expressions and other language constructs, however this is outwith the scope of this project.

A type signature may contain zero or more arguments, however there must be exactly one return type, if multiple values are returned this is done by constructing a tuple, $f : a_1 \dots a_n \rightarrow (r, r', r'')$. Type signatures are present in both of the following constructs, and examples of the syntax are seen in each line of the `Nat` definition above, along with the top line of the `add` definition and the `multiply` declaration.

2.1.2 Data Types

Data type declarations define a data type and describe how to construct it using data constructors. 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 keyword `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.

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

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

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

The `Nat` data type has two constructors `Zero` and `Suc`. Natural numbers are built up from zero by repeatedly adding one. To construct the number two, we call the Successor constructor twice, first with the argument `Zero`, and again with the result.

2.1.3 Function Definitions

The definition of `two` seen above is an example of a function definition, one which takes no arguments, and returns a value of type `Nat`. Functions have two components: A type signature, and a pattern matching definition.

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

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

Pattern matching 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 it is being applied to, 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 when matching does occur. Not all arguments that a function takes in must be listed on the left hand side, if certain parameters are left out, the return type will be of the form $p_1 \dots p_n \Rightarrow r$ where n is the number of remaining arguments, in this case a lambda expression can be constructed, taking in the remaining parameters. Lambda expressions have the syntax $\lambda x \Rightarrow y$, read “Lambda x returns y ”, where x is the argument term, and y is a term of the return type. In the example below, both lambda expressions on the right hand side have type `Nat -> Nat`.

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

2.1.4 Parametric Definitions

We define lists inductively, in the same fashion as natural numbers, by first building up from the base case, and successively 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 receives the parameter `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, as displayed in the example of the `map` function, the list may be transformed without any inspection of the elements themselves.

```
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 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. The following examples are introduced to display a small portion of the expressive power provided by dependent types.

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

The `mapAppendDistributive` type signature states that for any two lists, appending them together, then mapping a function over the result, is the same as mapping a function over each list individually, and appending the results. By constructing this equality, the `mapAppendDistributive` function may be used to satisfy the type checker where one side of the equality is required for a term to be well typed, but we can only construct the other.

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

Since division by zero is undefined, division may fail, the `Divides` view provides a way of describing integer division such that it is either division by zero, or a multiplication of the divisor and some number, plus a remainder. The resulting constructor can be inspected by the function using division, to ensure the undefined case is handled, avoiding potential run-time exceptions.

2.1.5 Differences from the Examples

For readability, the examples displayed have been presented in `Idris` code, not `TinyIdris` code, the differences between the two are discussed here.

```
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 differences that occur, each a result of `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, highlighted in green. The other difference is the patterns being matched must be passed in explicitly, requiring each name found in the application to be first brought into scope with a pattern variable, shown by the yellow highlights. More complete languages than `TinyIdris` also include 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 which can be one of three constructs: type signatures, data types, and function definitions. Type signatures (`IClaim`) consist of a name and a term describing the types for each input and the output type being constructed. 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
```

Expressions from the language are represented as the `RawImp` data type.

```
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
```

Each name referenced within a term will be done so via an `IVar` term, for example, `x`, `xs` and `append`. `Type` is the type of types, for example `Nat : Type`, and `Pi` types are the type of terms that take arguments, and can be read as saying "forall elements of the argument type, the return type holds". `add : Nat -> Nat -> Nat` would be parsed into pi binder, taking an argument of type `Nat`, and returning a pi binder which takes a `Nat` and returns a `Nat`. Within `TinyIdris Type`, the type of types, also has type `Type`, this can present issues via Girard's paradox [4], however this will not affect the results, as it can be difficult to accidentally come across the issues this presents accidentally.

`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 the arguments for that case, such as `append [] ys`. `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, in the definition for `add`, they occur both on the left hand side, after the pattern variables, `add (Suc n) m` and on the right hand side, applying the constructor `Suc` to `add n m`.

```
pat n : Nat, m : Nat =>
  add (Suc n) m = Suc (add n m)
```

`TinyIdris` has limited support of implicit arguments of the form `(x : _)` meaning the not all types must be stated explicitly, however each argument must be taken in explicitly, unlike the `Idris` code. `Implicits` may appear within type signatures where the type may be inferred, such as `append`.

```
append : (a : \_) -> (n : \_) -> (m : \_) ->
  Vec n a -> Vec m a -> Vec (add n m) a
```

The `ty` argument in `IPatvar` would be `Implicit` for the first three arguments of the `append` type signature when converted to `RawImp`.

`ILam` represents anonymous functions, that take an argument, 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 where not all patterns have been taken in on the left hand side and a function is being constructed, or where a function is being used as an argument to another function, for example `map (\ x => x + 1) xs` would map plus 1 over a list of natural numbers.

2.3 The Core Representation

After the source code has been parsed into a list of declarations, each declaration is then processed by the elaborator. 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 as a map of names to global definitions `GlobalDef`, 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, represented by the `Def` datatype.

Along with each of the definitions discussed earlier, `Defs` 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. Constructors are stored with a tag 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 representation are well scoped.

```
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
```

As an example of this well scoped property, when Processing an `IPi`, the name of the argument will be added to the list of names, therefore the list will increase in size with each binder passed. If one of the arguments taken in is `List a` then it will be represented as an application of an `IVar` referencing the name `List` to an `IVar` referencing the name `a`. `IVar` will be elaborated to either a `Ref`, when it is a reference to a global name, or a `Local` Where the name is in scope locally. If `a` is not a reference to a global name stored within the context, it must be in the scope locally, represented by the `Local` constructor¹. Construction of `Local` terms requires the index of `a` within the list of names, and a proof that it is valid. If `a` is not in the names, it will not be possible to construct the proof, and therefore the `Local`, and by extension, the application, are not possible to construct. If, on the other hand, it is within the list of names previously passed in, it is possible to construct the proof, and the resulting terms.

`IPi` represent one of four binder types, for convenience, all binders have been combined into the `Binder` data type, which can be used to construct a `Bind` term.

```
data Binder : Type -> Type where
  Lam : Name -> PiInfo -> ty -> Binder ty
  Pi : Name -> PiInfo -> ty -> Binder ty

  PVar : Name -> ty -> Binder ty -- pattern bound variables ...
  PVTy : ty -> Binder ty -- ... and their type
```

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

Pi and PVTy, both describe the types of terms being taken in, while Lam and PVar describe the terms being taken in. From the above example, the application of List to a would be wrapped in a Pi binder, which would be wrapped in a Bind term. The scope of Bind term is scoped by `x :: vars` whereas the argument is scoped by `textttvars`, meaning the name `x` being brought in for the argument may be referenced in the scope.

Considering the `isEmpty` function:

```
isEmpty : (a : Type) -> (xs : List a) -> Bool
pat a : Type, x : a, xs : List a =>
  isEmpty a (Cons a x xs) = False
pat a : Type =>
  isEmpty a (Nil a) = True
```

The type signature will parse to the RawImp term:

```
IPatvar a Type (IPatvar xs (IApp (IVar List) (IVar a) (IVar Bool)))
```

Which elaborates to:

```
Bind a TType (Bind xs (App (Ref nty List) (Local idx a) (Ref nty Bool)))
```

The pattern variables would parse and elaborate similarly. Moving through each pattern until the `=>` in syntax is reached, resulting in an application of the function name to the argument terms, for example `pat a : Type => isEmpty (Nil a)` would result in:

```
(IApp (IVar isEmpty) (IApp (IVar Nil) (IVar a)))
```

Which elaborates to:

```
(App (Ref nty isEmpty) (App (Ref nty Nil) (Local idx a)))
```

In this case the list is empty, therefore the syntax `True`, appearing on the right hand side of `=` would parse to `IVar True` which becomes `Ref nty True`.

Meta terms are constructed during unification, they have a name and a list of arguments to which they are applied, since they are only constructed during unification, they have no representation in RawImp or TinyIdris syntax. `App` terms represent an application of two terms, `TType` is the core representation of the `Type` RawImp term, and `Erased` are terms that have been erased during processing. 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. We shall only encounter environment populated by `Terms`. Since the system contains dependent types, terms may reference those previously brought into scope, the second argument is a list of names in scope, this ensures that if a term does reference an earlier term, the name is present in the environment. The data constructors enforce this; if the environment is empty, then there are no names in scope that can be referenced and each time a binder is added to the environment a name that may be referenced is added to the list of names in scope along 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 auxiliary data types.

```

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

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 un-evaluated. The outermost part, or head, is something that may be applied to arguments, such as a constructor or function, the head consists of a name being referenced, or a meta-variable created during unification.

The arguments are stored as ‘thunks’ within the `Closure` data type. They contain a term, and an environment 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` expressions to `Terms` in the core language.

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 of Idris there are several 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, proceeds by breaking down terms into their individual components, checking each component, which provides the appropriate `Term` and putting them back together as a `Term`. The

checkExp function may then be called with the expected value and the generated term. For example, when checking the term (**IApp** **f a**), the function is checked, if it is a pi binder, then the argument is checked, and checkExp is called with an App term, applying the checked function to the checked argument.

Alternatively, if the term being checked is a pi binder, then the argument is checked, and the environment extended with the resulting term, the scope of the binder is then checked within the updated environment. The checkExp function can then be called with a Bind using the resulting argument and scope terms, TType is used as the expected type.

If the given term is an **IVar** then the scope the name exists in is checked, resulting in a **Local** or **Ref** term which may be passed to the checkExp function.

The checkExp functions purpose is to check that the term generated matches the expected term, if there is no expected term then it succeeds automatically, 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 if successful, otherwise failing with a unification 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 identifying whether two terms may be substituted. It operates by receiving 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 convertible. 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 convertible, which checks of they are equal. Where subterms are unified, the union of the constraints generated is taken and returned.

If any of the terms within an application are meta-variables, the algorithm attempts to solve and instantiate the meta term, 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 during the process. Constraint solving is carried out at various points during type checking, including where holes have been solved.

3. Processing Declarations

The processing functionality follows a similar approach for each type of declaration. When processing type signatures for functions, type constructors and data constructors, type checking takes place with an empty local environment, this is possible since each argument must be passed explicitly. This results in a closed term (**Term []**) which may be added to the current context, as the type of the declaration.

The **ImpTy** case is straightforward, in the empty environment, the **RawImp** term is checked, with the expected type of **TType**. 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 as a **TCon**. Each data constructor is similarly checked

in the empty environment, and for each a new DCon definition is added with the resulting type. Each constructor has a tag, between 0 and the number of constructors, distinguishing them.

The most work is carried out 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 generated 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 constructed 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.

2.6 Extensions to the language

As part of this project the language has been extended with various features. User inserted holes of the form `?hole_name` have been added to the language, this requires a constructor `IHole : Name -> RawImp` within the `RawImp` data type, and the parser extended to accept the new syntax. The unification state has also been extended with a sorted map of Names to user holes, where IHoles are encountered during processing they are added to the map. The `Def` type has been extended to include the constructor:

```
MetaVar : (vars : List Name) -> Env Term vars ->
          (retTy : Term vars) -> Def
```

This represents user inserted holes. The `MetaVar` `Def` stores the local environment in which the meta-variable is defined, along with it's return type, and the list of names in scope where it is defined. The elaborator has also been extended to support user provided holes. Since the holes should appear on the right hand side of pattern matching definitions, there should always exist an expected type whenever one is encountered by the elaborator, which uses it, along with the current environment to generate a new `Meta` term using `MetaVar` as it's definition.

The definition for type constructors has also been altered to maintain a list of names referencing the data constructors that result in that type, this change has been propagated throughout the code where type constructor definitions are referenced.

To improve performance, certain other information has been added to the unification state. Two sorted maps of names, one to functions and one to types, have been added with the purpose of reducing the number of terms enumerated by the synthesiser. As the unification algorithm generates meta-variables, the size of the context quickly blows up with values which could not lead to a candidate term, therefore traversing the context itself for functions and types quickly becomes inefficient. The processing stages have 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 during synthesis the unification being performed is unsafe, since it is used to identify invalid terms, therefore the system it has been extended to handle failure of the unification algorithm. The unification algorithm itself has also extended to support the unification of binders, which is necessary for synthesis.

Certain exceptional behaviour may occur during synthesis that would not during the regular functioning of the language, such as the name being provided to the system not being found within the context, or having an invalid definition. The `Error` type has been extended to reflect this.

3 Related work

There have been several creative attempts at synthesising programs coming from many fields within computer science. Such as the machine learning, programming languages, and program verification communities. The machine learning research is outwith the scope of this project. 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. These developments have been shown to improve the performance of synthesis algorithms using a type driven approach. TinyIdris does not support quantitative types, therefore this research is outwith the scope of this project.

3.1 Automated Theorem Proving in Agda

Agda is a dependently typed programming language and interactive proof assistant. It is the closest relative to Idris, indeed the development of Agda heavily influenced that of Idris[3]. 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 number is a name inserted by the system to uniquely identify the 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 purposes. 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 non termination, 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. These are of the same form as a solution, except the term contains 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 previously 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 with new language 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[2].

The languages that Apsy and this project operate on have are very closely related, thus Apsy, and the tool developed for this project have a similar level of type information to them. The systems however, take very different approaches to synthesising programs, as Apsy utilises targeted knowledge, whereas this project uses a more general approach of type driven enumeration. Apsy has been designed to supplement the programmer during the software development process, interacting with the ide directly, thus does not attempt large definitions, instead timing out at 1000ms to avoid taking too long, this contrasts with the tool developed here which operates from the repl, and places no requirement on a time limit, instead using a depth limit.

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 relies on 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. As outlined in [5], an example of its operation is presented by the split function.

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

The intended solution should split on the list, returning a pair of empty lists if it is empty, a pair of an empty list and a singleton list if the list has one element, and if it has more than one element, making a recursive call, and adding the first element to the list in the first element in the resulting pair, and the second element to the list in the second element. This definition will synthesise an incorrect solution, simply returning a pair containing the initial list as it’s first and second elements. Allowing the programmer to refine the specification however can result 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)
}

def split(lst : List) : (List , List) = lst match {
  case Nil => (Nil, Nil)
  case Cons(h, Nil) => (Nil, Cons(h Nil))
  case Cons(h1, Cons(h2, t2)) =>
    val r = split(t2)
    (Cons(h1, r._1), Cons(h2, r._2))
}
```

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 where 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 a formula in first order logic, on which satisfiability checking can be performed by a solver. The Leon system uses the Z3 solver, which checks if there is any instantiation of the formulas variables such that the entire formula is true. This formula is passed into a *refinement loop*. Refinement loops search for values satisfying the condition where the synthesis predicate is true, this is restricted via iterative deepening to avoid termination issues. 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. 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.

The type system present in the Scala programming language is weaker than that of dependently typed programming languages, however the extension of the language with symbolic descriptions, combined with the Z3 solver, provides a powerful way to express properties of the target terms. This could prove to be a useful extension to the TinyIdris system, increasing the expressiveness of the language. The tools ability to stop mid search and provide a best effort definition is a useful addition, which also provides inspiration for potential future work.

3.3 Type and Example Directed Program Synthesis

The Myth system[10] treats program synthesis as a proof search, using 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’ with each input/output pair existing 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 un-decidability 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, performing E-guessing at each node, and pushing 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.

As described by the creators of Myth [10], it was tested on a set of problems surrounding the data structures, Boolean’s, 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.

This project implemented in TinyIdris does not currently utilise the IRefinement steps found within the Myth system, however doing so may provide significant performance enhancements, however experimentation would be required to verify this in practice. The use of input / output examples however adds a significant burden on the programmer, which limits the practicality of Myth, suggesting that the use of a stronger type system provides a more useful tool for aiding programmers.

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 output type appears before the `|`, and the condition appears after it, in Synquid, the output is referred to by the variable `v` within the condition. In this example, the output `List A`, has been refined by the condition that the length of the output 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, first used withing the Leon system. 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 have been 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 succinctly, but to significantly improve performance.

```

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 system still suffers from downsides, as displayed in the above example, specifications can still become large, 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.

Synquid, by utilising refinement types and satisfiability solving allows specifications to be expressed concisely in a way that cannot be done using only dependent types. Synquid also requires extra information such as measures and termination measures, limiting the benefits gained in concise descriptions. Both Synquid and the tool presented here operate in a similar way, when provided with a source file and a name that is to be defined, it attempts synthesis, and presenting the resulting code to the user, to be inserted into the code, however, the language that Synquid can accept and synthesise has been developed only for the system, therefore cannot be executed, which limits testing, and the usefulness of the tool.

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 contains considerably more language features, and has much more information available due to the more sophisticated type system. The algorithm follows certain steps: When provided with a hole, first 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 detailed as 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.

Synthesis functionality present within the Idris2 system is the closest relation to the tool presented for this project, with both attempting to synthesise full pattern matching definitions on extremely similar languages, however Idris2 has a more sophisticated type system, which it can utilise, increasing the effectiveness of the tool. The approach taken to synthesis is similar, however the Idris2 does not attempt to use function definitions to synthesise terms, instead focusing in on recursive functions only.

The Idris2 tool is built directly into the ide, as such it places more importance on performance than the tool presented here, attempting synthesis in batches, and cutting off after a certain number of terms have been synthesised, or a time limit has been reached. The Idris2 tool also offers a more interactive experience, allowing the user to loop through each definition returned, allowing them to select the correct one if the ordering heuristic does not select it.

Certain problems encountered during the synthesis of terms are distinct to each language, with the lack pattern variables in the full Idris2 implementation bypassing certain issues, and the simplified system of TinyIdris similarly avoiding issues present within the full implementation.

4 The Synthesis Tool

The program synthesis functionality implemented in this project is outlined here. To allow user interaction, the TinyIdris repl has been extended with an command, **auto**, which takes the name of a hole, or a type signature with no definition. Depending on the input, the tool will move into one of two modes of operation; Synthesising an individual term, or a full pattern matching definition, the process of each is outlined in figures 4 & 5. The tool returns a string of TinyIdris syntax, which can be inserted into the source file, otherwise an error is returned if synthesis fails. An example of the tool in operation is displayed in figure 3.

```
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
> █
```

Figure 3: Synthesising a definition, term, and failing on a term.

Figure 4 outlines the process for synthesising individual terms. To support this, the language has been extended with ‘holes’ of the form `?hole_name`. When the name of a hole is provided to the tool, it attempts to synthesise a single term. The process followed initially attempts to synthesise terms using variables in scope locally, followed by using data constructors, finally functions from the context are attempted. Each way of generating terms will return a list of candidate terms to be ordered by a heuristic. The resulting term is then un-elaborated to RawImp, which can be re-sugared into TinyIdris syntax.

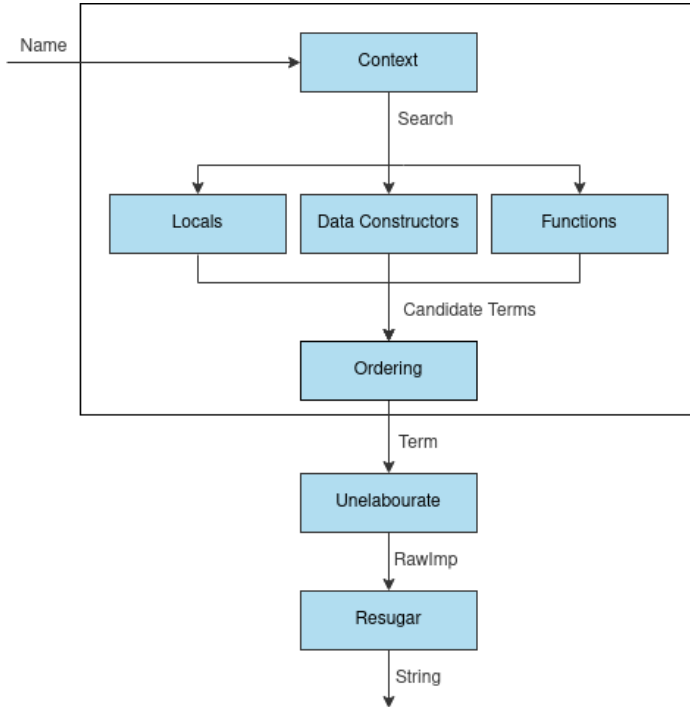


Figure 4: Synthesising Terms

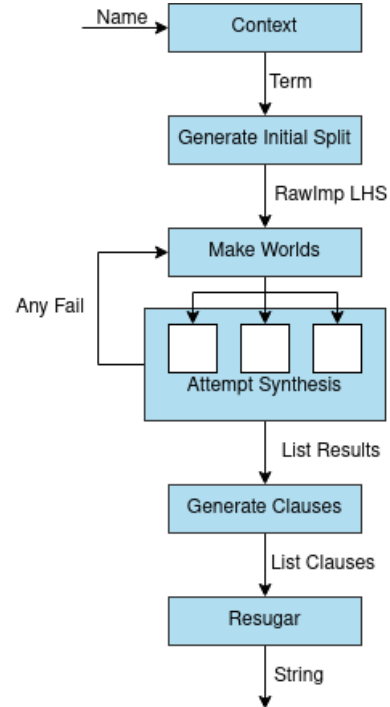


Figure 5: Synthesising Definitions

The process for synthesising pattern matching definitions is outlined in figure 5, and is the more complex of the two modes of operation. When provided with a type signature, it begins by generating an initial left hand side `RawImp` term, in which no pattern matching occurs. Case splitting is then performed on the left hand side, and for each case generated, synthesis is attempted utilising the process for individual terms, if any cases fail, the process returns to the case splitting stage and the process repeats until a each clause may be synthesised, or no more case splits may occur. If each clause may be synthesised, the results are combined into a list of clauses, which may be re-sugared into a string of `TinyIdris` syntax.

4.1 Synthesising Individual Terms

A `Search` is represented internally as:

- A `Nat` depth limit.
- The `Name` for the expression being synthesised.
- The `Env` Local environment.
- The `RawImp` left hand side of the target term.
- A `Term vars` target type.

The depth is introduced to avoid termination issues which are presented by the depth first nature of the search, for example when synthesising a term of type `Nat` and attempting to use the `Suc` constructor, which takes a `Nat` argument, the tool would attempt to synthesise an argument of type `Nat`, and so on. An initial depth of 4 has been found to be sufficiently deep to provide useful results, without incurring a large performance cost.

The name, environment 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 when their type is actually a type, and not any other expression from the language. For example if the type is a pi binder then we may construct a lambda bringing in something of the argument type, and attempting to synthesise the scope, however, if the type being attempted is a lambda, we have no way of constructing something of that type. Similarly, if the term at the end of each pi binder is a reference to a type constructor we may proceed, however if it is a data constructor, we cannot.

If the target type is of type `Type` then we are able to synthesise valid terms by using anything of type `Type` in the context, such as `Nat`, `List Nat`, `Bool`. Following this approach will lead to several incorrect answers being attempted, since synthesis of types will typically occur while synthesising arguments for a definition, which are not typically random types from the context. As `TinyIdris` requires that arguments are passed in explicitly, the types present in the local scope will most likely be the desired ones, for example when attempting the `Cons` constructor for the `append` function, the first argument on the left hand side will be the type of the lists passed in, as well as the first argument to the resulting list constructor on the right hand side. For this reason, the types attempted are restricted to those in scope locally.

If the target term is a pi binder we may construct a lambda, taking an argument of the required type, and attempting synthesis on the scope. After moving through each pi binder, the resulting scope must

be an application of a type constructor to zero or more arguments, for example `Nat`, `List a`, `Vec n a`. 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 only, which will require a maximum of two passes of the environment, before terminating. Otherwise the system performs a full search.

A full search is carried out first by checking the variables in scope locally, followed by attempting to use the data constructors for the target type, followed by function definitions which have the desired return type. If a valid term is found, but has arguments to which it must be applied, a depth first traversal of the arguments takes place, attempting to synthesise each argument, until one fails, more information generated shows that the term cannot be valid, or all arguments have been synthesised, to which the term may be applied. A list of potential candidates will be returned, which are ordered based on heuristics.

4.1.1 Searching Locals

The first terms attempted by the algorithm are those present in the local environment. These are commonly the desired terms when defining the base case of a recursive function, as displayed in the `Nil` case of the `append` function below, or when synthesising arguments for a top level term being synthesised, such as `x`, `xs` and `ys` in the `Cons` case.

```
append : List a -> List a -> List a
append [] ys = ys
append (x :: xs) ys = Cons x (append xs ys)
```

The process is split into two stages of searching for valid local variables is split into two stages. Usable terms must first be retrieved from the environment, those found may then be checked for validity.

`Pi` and `PVTy` both describe the types of bound terms, whereas `PVar` and `Lam` represent the terms themselves. 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 usable then we must construct a `Local` term referencing the name, for example `pat n : Nat =>` would then result in `Local 0 n`, taking care to update indices as each binder is passed. The resulting list of local variables is then passed to the second stage, which traverses the list, getting the binder `Type` from the environment for the local variable, in the above example this would be `Nat`, and attempting unification between this and the target. If no constraints are generated then the `Local` term is accepted, and the rest of the environment is checked. In this step we are looking for a direct match, therefore any terms which require constraints to hold should be disregarded, since we have no guarantee that they do.

To ensure the tool is able to apply higher order functions, the binder type is checked if unification with the target fails. If it is a function type, e.g `f : (a -> b)`, the arguments are filled by meta-variables and unification is once again attempted with the target type. If this is successful then synthesis of each of the functions arguments are attempted, in this example a term of type `a`, and if successful an application of the function to the synthesised arguments is returned.

4.1.2 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 $p_1 \rightarrow \dots \rightarrow p_n \rightarrow r$ where $p_1 \dots p_n$ are arguments to the function and r is an application of a type constructor to zero or more arguments. For example, when attempting to synthesise a non empty vector, the Cons constructor would be attempted, which would be represented as:

`(Cons, Data Constructor, (a : Type -> n : Nat -> x : a -> xs : Vec n a -> Vec (S n) a))`

The process for attempting to use a function definition is the same as functions in scope locally, meta-variables are constructed for each binder, in this case a , n , x and xs , resulting in the term $\text{Vec } (S \ n) \ a$. 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. If at this stage unification produces constraints then synthesis is continued, as it would be expected that constraints would be placed on each of the meta-variables. A potential optimisation of the system would use this information to search more efficiently, such as the *I-Refinement* rules found in the Myth system. Unification returns constraints only on the return types, therefore constraints of the form (**meta-variable** = **term**) are not typically generated. Potential enhancements could use information about which arguments are present within the return type of constructors to identify these constraints, or make use of SAT solvers as seen in previous systems.

When synthesising arguments, it is possible that arguments deeper into the binder may depend on arguments that have been previously passed in, in Cons, the argument x depends on the first argument a , and xs , depends on the first two arguments. As a result, branching occurs as the traversal moves down through each 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 unification 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, with the resulting term unifying with the target type, at which point a Reference to the name of the definition being attempted is returned, to which the synthesised terms may be applied. To ensure that arguments are applied in the correct order, they are pushed to the front of the application, as displayed in figure 5.

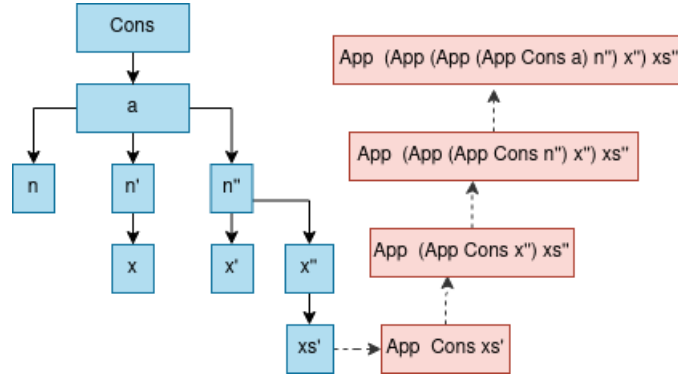


Figure 6: Synthesising an application.

4.1.3 Structural Recursion Checking

TinyIdris does not have any termination checking built in, therefore care must be taken to ensure that where a recursive definition is synthesised, checks have been put in place to ensure there is at least one base case, and that the terms generated on the right hand side are structurally different to those on the left hand side, preventing the function from recursing forever.

The structural recursion check takes as an input a left hand side term and a right hand side term. The check is conservative, in that it does not reduce terms, so may deny terms which are structurally different, for example `append xs ys = append xs ys` would be correctly stopped, however `append xs ys = append xs (drop 1 ys)` would also be rejected, even though it is 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, such as `append xs ys = append ys xs`.

The base case check is straightforward, it takes a name and list of right hand side clauses, and checks that at least one of the clauses does not make a recursive call. Although this will block functions from recursing forever, it makes no attempt to block mutually recursive functions from being defined, in the following example, the top function would be rejected, however the bottom two would not.

```
caught : List a -> List a -> List a
caught (x :: xs) ys = caught xs ys
caught [] ys       = caught [] ys

uncaught : List a -> List a -> List a
uncaught xs ys = badRecursion xs ys

badRecursion : List a -> List a -> List a
badRecursion xs ys = uncaught xs ys
```

4.2 Synthesising Definitions

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

The only information available 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.

4.2.1 Pattern Matching

The splitting algorithm receives either a singleton list containing the initial left hand side, or a list of multiple left hand side `RawImp` terms that have been generated by a previous split, each of which is split again. After an initial split takes place, each left hand side term will have at least one unique argument, which will not be split on, therefore each generated left hand side will remain unique. The type checker is used to filter out any invalid terms generated at this stage.

The terms split on initially will be the leftmost type constructor. However, following a strict left to right split will lead to multiple redundant splits, this problem is exacerbated since each argument must be passed explicitly. To reduce this, a traversal of the generated split occurs, which uses the type information available from the initial split to fill in any implicit arguments generated, if the new information provided by the split results in any deeper terms being forced into a certain pattern, then that argument is replaced by the pattern. Again the resulting left hand sides are checked by the type checker to ensure their correctness. This traversal is repeated each time a split is generated, either from an initial split or a forced split.

After new splits have been generated each is split into its own world, in which synthesis is attempted. If each world results in a valid term, then combining the worlds into a definition is attempted, if this results in a valid definition, it is accepted, otherwise each world is further split. A potential future optimisation could only split the worlds for which synthesis is unsuccessful.

For each world to be synthesised the `RawImp` left hand side must be converted into an environment and a term, a list of candidate terms may then be synthesised for the target. Synthesis is attempted for

each remaining world, if any fail then the process is killed, returning to the splitting stage, otherwise the top term synthesised is taken, to avoid a major increase in the number of definitions for which synthesis is attempted. The selected term is converted into a clause, which is returned along with the rest of the synthesised clauses.

4.2.2 Combining Clauses

TinyIdris has a built in `CaseTree` representation for clauses, when the synthesiser returns a set of clauses, it is combined into a case tree, 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. Structural recursion checking is performed at this step, to ensure that there is at least one base case, and each right hand side term is structurally different to the left hand side term. After a valid term has been synthesised, be it an individual term or full pattern matching definition, the result is converted into a string of TinyIdris syntax for the user to insert into the source file.

5 Detailed Implementation

5.1 Synthesis

Searches within the system are defined as a record type:

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

  run : {auto c : Ref Ctxt Defs} ->
        {auto u : Ref UST UState} ->
        Name -> Core String
```

While this is not necessary for the functionality, it serves to clean up the type signatures of functions, improving readability. The majority of the functionality is wrapped up in the `Core` monad, which is a major part of the TinyIdris system, the Monad is a wrapper around the IO Monad, which allows IO operations to be carried out, `Core` extends this functionality to include global references for the context and unification state. Since these are required frequently, much of the synthesis functionality must be wrapped inside `Core`.

When a search begins, the `run` function, with the type signature above is called, this has the purpose of checking which kind of search is being attempted by checking the definition from the context, running the search, and converting the result back into a string via the re-sugaring functionality. In the event that the search is an individual term, the results are filtered here to ensure that they are structurally different from the left hand side. This check occurs earlier for pattern matching definitions.

5.1.1 Individual Terms

If the definition from the context of the name for which synthesis is to be attempted is a `MetaVar`, the left hand side of the clause the hole is defined in is retrieved from the userholes map within the unification state, together with the environment and return type from the `Def`, and the name, this is combined to generate a search, on which the synthesise function is called.

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

The synthesise function, when given a search will return a list of candidate terms. The function operates by matching on the return type, if it is a pi binder then the function is called recursively with an extended environment taking in a lambda, and the scope. If the target is of type `Type`, or the depth is zero then it searches the environment, otherwise it performs a full search.

A full search is performed by trying local variables, then data constructors, and finally function definitions.

When searching local variables, we must be able to construct `Local` terms referencing them, to do this the environment must be 're scoped', this is handled by the `getUsableEnv` function.

```
getUsableEnv : {vars : _} ->
  (ns : List Name) ->
  Env Term vars ->
  List (Term (ns ++ vars))
getUsableEnv {vars = v :: vars} ns ((PVar x z) :: env)
= let rest = getUsableEnv {vars = vars} (ns ++ [v]) env
  MkVar var = weakenNS ns (MkVar First) in
  Local _ var :: rewrite appendAssociative ns [v] vars in rest
```

Re-scoping the environment consists of checking which terms from the environment are of a usable form, either lambdas or pattern variables. The term representing the type from the binder is then weakened using the list of names that are passed in. Weakening is simply extending the list of names that are in scope for a given term. The function returns a local term referencing the term in the environment. A proof of associativity is required to satisfy the type checker that the list of names from the recursive call is correct. By calling the `getUsableEnv` function with the empty list of names, the returned terms will be in the scope `[] ++ vars` which is equal to `vars`, the original scope.

Searching these terms consists of looking up the binder within the environment, and attempting to unify its type with the return type. If this is successful the local is returned, and the search continues with the rest, if this is unsuccessful, and the depth is greater than 0 then the type of the binder is checked in case it is a function with the correct return type. If it is then synthesis is attempted for the arguments, otherwise the search continues with the rest of the locals.

In order to check if the result of a pi binder is the type we are looking for, the binder must be normalised, which can then be filled with meta-variables, and quoted back to a term. Unification can then be attempted between the return type and the filled type, which will either succeed, potentially with constraints or fail.

```
fillMetas : {vars : _} ->
  {auto c : Ref Ctxt Defs} ->
  {auto u : Ref UST UState} ->
  Env Term vars -> NF vars ->
  Core (NF vars , List (Term vars, Name))
fillMetas env (NBind n (Pi n' pinfo tm) scope)
  = do defs <- get Ctxt
      nm <- genName "filling"
      mta <- newMeta env nm !(quote defs env tm) Hole
      (f , args) <- fillMetas env !(scope defs (toClosure env mta))
      pure (f , (mta , nm) :: args)
fillMetas env tm = pure (tm , [])
```

The task of filling terms with meta-variables is carried out by the `fillMetas` function, which takes an environment and a value scoped by this environment. A meta-variable for the argument type is generated and added to the context, this term may then be converted to a closure and passed into the scope of the bind value, with which a recursive call is made. After each recursive call is made, the function will return the term at the end of the scope and a list of the meta-variables generated along with their names are returned for access.

The process for determining the validity of local functions is the same as that taken for data constructors and function definitions. The process outlined above is handled by the `tryDef` function for global definitions, which operates in similarly. The `synthesise` function determines the correct data constructors to attempt and accesses the defined functions from the unification state, calling the `tryDef` function for each, and concatenating the results.

Once it is known that a function type may return a valid candidate term, synthesis of each argument is attempted by the `tryIfSuccessful` function.

```

tryIfSuccessful : {vars : _} ->
  {auto c : Ref Ctxt Defs} ->
  {auto u : Ref UST UState} ->
  (Search vars) ->
  Name -> NameType ->
  NF vars -> Core (List (Term vars))
tryIfSuccessful s@(MkSearch (S depth) name env lhs target) n nty (NBind m (Pi nm pinfo tm) sc)
  = do defs <- get Ctxt
    (tm' :: ts) <- synthesise (MkSearch (pred depth) m env lhs !(quote defs env tm))
    | _ => none
    pure $ concat !(traverse help (tm' :: ts))

where pushdown : Term vars -> Term vars -> Term vars
      pushdown tm (Ref nty n) = App (Ref nty n) tm
      pushdown tm (App f a) = App (pushdown tm f) a
      pushdown tm f = App f tm

      help : Term vars ->
        Core (List (Term vars))
      help tm
        = do defs <- get Ctxt
          sc' <- sc defs (toClosure env tm)
          (filled , fas) <- fillMetas env sc'
          Just ures <- tryUnify env target !(quote defs env filled)
          | _ => none
          (r :: rs) <- tryIfSuccessful s n nty sc'
          | _ => none
          pure $ map (pushdown tm) (r :: rs)
tryIfSuccessful (MkSearch 0 name env lhs target) n nty tm = none
tryIfSuccessful (MkSearch depth name env lhs target) n nty tm
  = do defs <- get Ctxt
    Just (MkUnifyResult [] holesSolved) <- tryUnify
      env
      target
      !(quote defs env tm)
    | _ => none
    case nty of
      Bound => case defined n env of
        Nothing => throw (GenericMsg "Bound not in env")
        (Just (MkIsDefined p)) => pure $ [Local _ p]
      _ => pure $ [Ref nty n]

```

Synthesis of function arguments arguments involve the handling of branching, which must occur since each argument to be synthesised may depend on the arguments previously synthesised in a given branch. The function operates by synthesising a term for the first argument, when this takes place the depth is reduced to ensure termination of the program. A new branch is created for each of the synthesised terms. Each branch is carried out by the helper function ‘help’ which attempts unification, and continues down the branch if successful, or killing the branch if not. The synthesised argument must be passed as a closure into the value in order to evaluate the scope of the binder. When every argument has been synthesised, a list of the synthesised arguments is returned to which the function will be applied.

5.1.2 Pattern Matching Definitions

The generation of pattern matching definitions can be split into three main problems, splitting the left hand side, handling synthesis of multiple cases, and combining cases. All of which is handled by the `defSearch` function.

```

defSearch : {auto c : Ref Ctxt Defs} ->
  {auto u : Ref UST UState} ->
  GlobalDef -> Name -> List RawImp -> Nat ->
  Core (Maybe (Def, List (Clause, RawImp, RawImp)))
defSearch def n lhs splits =
  do cs@(c :: cases) <- traverse (\ l => splitLhs False splits [] l) lhs
  | _ => pure Nothing
  gs@(p :: ps) <- filterCheckable (map (\ a => (a,[])) (concat cs))
  | _ => pure Nothing
  Just res <- synthesisePM n (type def)
    !(traverse (\ (_,gd,ri,_) =>
      pure $ getSearchData !(getTerm gd) [] ri) gs)
  | _ => defSearch def n (map (\ (_,_,ri,_) => ri) gs) (S splits)
  pure $ Just res

```

The tool, when given a single left hand side, performs a split initially, this is a heuristic to help prevent against correct, but undesirable terms. For certain types, for example, `Maybe`, the tool will always be able to synthesise trivially true terms, `Nothing`, for `Maybe`, which is unlikely to be the desired solution for each case.

To ensure that synthesis is only performed on correct terms, any invalid terms generated are filtered out by the type checker. This approach leads to redundant splits, therefore, each argument that depends on the split argument is checked in case it has been forced into being constructed by a single constructor, if it has, then it is also split, this helps to reduce the number of cases generated. For example, an initial split on `n : Nat` being split to `Z`, the `Cons` constructor will no longer apply for any vectors of length `n`, thus the vector split to `Nil` would be accepted.

If a set of splits has been generated, the synthesis is attempted, otherwise the algorithm fails. Generating splits will be covered in more detail in the next section.

Attempting synthesis on a set of left hand sides is handled by the `synthesiseWorlds` function, Each left hand side is split into its own world. The algorithm recursively loops through each world attempting to synthesise a term of the correct return type. If any fail the process is stopped. If synthesis for each world is successful then it is checked that each clause contains at least one structurally different argument when making a recursive call. If the checks pass, to avoid greatly increasing the number of cases synthesis is being attempted for, the first synthesised term is selected and the resulting information is combined to a clause, which is returned along with the clauses generated from the recursive call. The process for generating each clause from the synthesised term is straightforward, checking the the left hand side, and un-elaborating the right hand side.

```

synthesiseWorlds : {auto c : Ref Ctxt Defs} ->
  {auto u : Ref UST UState} ->
  Name ->
  List (vars ** (Env Term vars , Term vars, RawImp)) ->
  Core (Maybe (List (Clause, RawImp, RawImp)))
synthesiseWorlds n [] = pure $ Just []
synthesiseWorlds n ((vars ** (env, tm, ri)) :: xs)
  = do let s = (MkSearch 4 n env ri tm)
      (t :: ts) <- synthesise s | _ => nothing
      (t' :: ts') <- filterM (structuralRecursionCheck env ri) (t :: ts) | _ => nothing
      Just rest <- synthesiseWorlds n xs | _ => nothing
      clause <- getClause (s, t', ri)
      pure $ Just (clause :: rest)

```


5.1.3 Pattern matching

Pattern matching occurs in two forms, when generating a split given a left hand side, and refining a left hand side which has been split previously, and contains arguments that depend on the split argument.

It is only type constructors that may be split on currently, however a potential extension could look to split type constructors that have been applied to data constructors on the left hand side.

The initial case simply traverses the term left to right, looking for the first type constructor which may be split into a set of data constructors. The second case, which occurs after the initial split has taken place, uses the list of names returned from the initial split to test which arguments may be refined.

The entire process is undertaken by the `splitLhs` function, which finds the initial term, splitting it into the new terms using the `getConPatterns` function, and using the `splitSingles` function to carry out the second stage.

```

splitLhs : {vars : _} ->
  {auto c : Ref Ctxt Defs} ->
  {auto u : Ref UST UState} ->
  Bool -> Nat -> Env Term vars -> RawImp ->
  Core (List RawImp)
splitLhs in_split splits env (IPatvar x ty scope)
  = do defs <- get Ctxt
      (tm , gtm) <- checkTerm env ty Nothing
      let (IVar n) = getFn ty | _ => cont x tm
      Just def <- lookupDef n defs | _ => cont x tm
      let (TCon tag arity datas) = definition def | _ => cont x tm
      (NTCon _ _ _ cs) <- nf defs env tm
      | _ => throw (GenericMsg "TCon not normalising to NTCon")
      spt <- traverse
        (getConPatterns splits (IPatvar x ty scope) env cs x) datas
      finals <- traverse (splitSingles (S splits)) spt
      pure $ concat finals
where
  cont : Name -> Term vars -> Core (List RawImp)
  cont x tm = if in_split
    then pure []
    else
      splitLhs {vars = (x :: vars)}
        in_split
        splits
        (PVar x tm :: env)
        scope
splitLhs _ _ _ _ = none

```

```

getRaw : {vars : _} ->
  {auto c : Ref Ctxt Defs} ->
  {auto u : Ref UST UState} ->
  Nat -> Nat -> Env Term vars -> List RawImp -> Name ->
  (List (Closure vars, Closure vars), NF vars) ->
  Core RawImp
getRaw splits depth env rs dcn (cs, (NBind y z f)) =
  do defs <- get Ctxt
  qtz <- quote defs env (binderType z)
  f' <- (f defs (toClosure env qtz))
  case lookupClos cs y of
    Nothing =>
      do nm <- genName $ "ic" ++ (show splits) ++ (show depth)
      pure $
        IPatvar nm Implicit
        !(getRaw splits
            (S depth)
            env
            (IVar nm :: rs)
            dcn
            (cs, f'))
      Just (x, clo@(MkClosure _ _ tm)) =>
        getRaw splits depth env (x :: rs) dcn (cs, !(f defs clo))
getRaw splits depth env rs dcn (x, tm)
  = pure $ apply (IVar dcn) (reverse rs)

getConPatterns : {vars : _} ->
  {auto c : Ref Ctxt Defs} ->
  {auto u : Ref UST UState} ->
  Nat -> RawImp -> Env Term vars ->
  List (Closure vars) -> Name ->
  Name -> Core (RawImp, List Name)
getConPatterns splits ri env cs name n
  = do defs <- get Ctxt
      Just def <- lookupDef n defs
      | _ => throw (NotInContext n)
  let (DCon t a) = definition def
      | _ => throw (InvalidDefinition n)
  norm <- nf defs env (rewrite sym (appendNilRightNeutral vars)
                              in weakenNs vars $ type def)
  dst <- getSplitData norm env cs norm
  raw <- getRaw splits 0 env [] n dst
  let (pats, app) = splitPats raw
  let (front, back, names)
    = case splitOnN ri name of
      (Nothing, suf) =>
        ((envToPat env) . pats, fixUpScope name suf app)
      ((Just pre), suf) =>
        ((envToPat env) . pre . pats, fixUpScope name suf app)
  newtm <- fillImplicits (front back) [] (front back)
  pure (newtm, names)

```

When the `getConPatterns` function is called, it is provided with the list of closures found in the value of the type constructor being split, this provides the tool with information on certain arguments passed in to the constructor, allowing certain implicit arguments to be filled. This is handled by the `getSplitData` and `getRaw` functions. The functions normalise and traverse the data constructor type, to reach the return type, which should be the same as the original type constructor being split, with different arguments in the closure. The list of closures present in this value may then be zipped with

those from the original type constructor, at which point a second traversal of the value occurs. The purpose of this is to check if any of the arguments being taken in are present within the list of closures of the evaluated data constructor. If they are, no pattern variable is required to take in the argument, as the value must be that of the original type constructor, which must already exist in the scope, and to which we may apply the constructor. If an argument is not present within the closure then a pattern variable is generated with a unique name, taking in an implicit argument, which the constructor is applied to in. This results in a `RawImp` for the pattern variables, returning an application of the data constructor to its arguments.

As an example of this, where the desired type is `Vec 4 Bool`, the `Cons` constructor will be attempted. The list of closures from the initial type will contain `[4, Bool]`, when the `cons` constructor is normalised, the resulting value will be `Vec n a` with the closures `[n, a]`, the zip results in `[(n, 4), (a, Bool)]`. When the second traversal, the arguments `a` and `n` will be filled by the values `4` and `Bool`, however the remaining two arguments, `x` and `xs` must be filled in by implicit arguments.

In the `getConPatterns` function, the result may be split into two, the new patterns required and the application to replace the original type constructor with. The environment is used to construct a term taking in each pattern variable that occurs before the split term, which is then merged with the new patterns. The original `RawImp` term after the name appears is then altered, replacing each occurrence of the name with the new application. Each part is combined generating a valid `RawImp` Term.

Before continuing with the second stage of the process the algorithm attempts to use any information it has generated during the split to fill in any `Implicit` arguments, via the `fillImplicits` function.

```
fillImplicits : {vars : _} ->
  {auto c : Ref Ctxt Defs} ->
  {auto u : Ref UST UState} ->
  RawImp -> Env Term vars -> RawImp ->
  Core RawImp
fillImplicits (IPatvar x Implicit scope) env ri
  = do (tm, gd) <- checkTerm env Implicit (Just gType)
      let env' : Env Term (x :: vars)
          = (Lam x Explicit tm :: env)
      fillImplicits scope env' ri

fillImplicits (IPatvar x ty scope) env ri
  = do (tm, gd) <- checkTerm env ty Nothing
      let env' : Env Term (x :: vars)
          = (Lam x Explicit tm) :: env
      fillImplicits scope env' ri

fillImplicits (IApp f a) env ri
  = do (ftm, gfty) <- checkTerm env f Nothing
      fty <- getNF gfty
      defs <- get Ctxt
      ri' <- fillImplicits f env ri
      case fty of
        (NBind x (Pi _ _ ty) sc) =>
          do
            (atm, gaty) <- checkTerm env a
                                (Just (glueBack defs env ty))
            case a of
              (IVar y) => pure $ replaceImplicitAt y !(getTerm gaty) ri'
              (IApp y z) => fillImplicits (IApp y z) env ri'
              _ => pure ri'
            t => throw (GenericMsg "Not a function type")
      fillImplicits tm env ri = pure ri
```

The `fillMetas` function repeatedly accepts arguments, checking their term and extending the environment with them. When it reaches the application at the end of the term it progressively moves through the arguments left to right, at which point it may have enough information to fill in the type of

any implicit arguments with their expected type, if so the implicit argument is replaced by the required RawImp term, if this is not possible, the argument remains implicit.

Along with the resulting RawImp term, each split generates a list of names which depend on the split argument, the second stage of the process follows a similar pattern as outlined above, however traversing the list of names, attempting to split each, as opposed to searching for the first type constructor. This is carried out until there are no names left that depend on any split arguments, at which point the resulting RawImp terms are returned.

5.1.4 Un-Elaborating and Re-Sugaring

The un-elaboration and re-sugaring algorithm are straightforward, converting a terms from TT back into RawImp and RawImp to syntax respectively.

When re-sugaring, the syntax of expressions will depend on it's location within a term, to handle this, helper functions have been created to distinguish the cases. Re-Sugaring full definitions is split into two stages, re-sugaring the type signature, and re-sugaring the list of clauses. Re-sugaring clauses simply re-sugars the left and right hand side of each, combining them with an equals sign.

6 Evaluation

6.1 Test Suite

Testing takes the form of running the tool on various function type signatures and evaluating the correctness of the resulting definitions. The result will be considered correct if on every possible input, it will return the correct output.

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, Agda’s auto search tool, or that used to evaluate the Synquid system. The categories are:

- Lists
- Vectors
- Vectors with proofs
- Equalities

The test suite has been designed to test the tool with varying levels of type information. As the level of type information increases, the number of possible incorrect definitions decreases, however the definitions themselves become more challenging for the system to synthesise.

The list tests will indicate the effectiveness of the tools ability to synthesise terms without relying on detailed type information. Vectors contain the same examples as lists, with the added type information of the length built in, it is expected that this will reduce the number of trivial terms synthesised, without providing detailed type information. Where the added type information present in vectors is not enough for a successful attempt, more type information is added to reduce the number of possible terms. Where applicable, synthesis will be attempted with types that not only return the result, but a proof that the result satisfies some property defining it’s correctness.

Testing Equalities focuses on the effectiveness of pattern matching as the results depend on information gained from splitting on equality proofs provided as arguments.

6.2 Testing Functionality

The system has been extended with functionality for carrying out testing of the synthesis tool. Tests may be carried out manually, by calling the tool and inspecting the result, alternatively tests can be carried out individually or in batches using a pre-defined answer file. These modes of operation are only available when testing individual terms, not full pattern matching definitions.

Testing of an individual hole can be carried out using the command `t` followed by the name of the hole. The system will run the tool and compare the output string to that in the answer file. The tool returns a printout, containing the hole name, expected result, actual result and ‘pass’ or ‘fail’.

The command `test` can be used to test an entire file. The command takes no arguments, instead attempting to synthesise a term for each name found in the answer file. For each name, the tool returns the printout as above along with a success rate over all the tests, an example output is displayed in figure 7.

```
> 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
> █
```

Figure 7: Batch and Individual Testing

6.3 Testing Plan

To test the tools, each test has been converted into TinyIdris, Idris, Idris2, Agda and Synquid code. To ensure that each tool has the same level of information available to it, all data types and functions have been created within the test files, ensuring no information from built in libraries may be used.

Although care has been taken to ensure the consistency of available information, fundamental differences between the languages do occur. Each tool being tested operates on a language that supports dependent types with the exception of Synquid, which instead uses ‘Liquid refinement types’. Where applicable, tests have been converted to the different type system, however, this is not possible for all tests. Where tests are not able to be expressed within the Synquid system, the result set contains the character ‘-’.

Similarly, with the exception of Idris, each system supports synthesis of pattern matching definitions. When testing was carried out on the Idris system, tests were first attempted without manual case splitting, where the tool failed, tests were repeated after manual case splitting. Splitting was carried in the order most likely to present a result, since each systems process for case splitting may be different. This manual splitting should not provide any advantage to Idris as with enough splits every other system has the opportunity to learn the information provided by the case split, depending on the implementation of case splitting however, some systems may not reach this point.

When evaluating the performance, we begin by looking at the tools results in isolation, and follow up with a comparison to other systems, along with alterations to the system.

6.4 Performance

Failed tests within the following tables are left blank, success is represented by ‘Pass’, and tests not expressible within a certain language are denoted by ‘-’.

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

Table 1: Lists

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

Table 2: Vectors

Problem	TinyIdris	Idris	Idris2	Agda
is empty	Pass		Pass	
is elem				
duplicate			Pass	
zip			Pass	
i'th elem				
index				

Table 3: Vectors with Proof

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

Table 4: Equalities

6.5 Evaluation of performance

As displayed in table 1, on lists, the overall performance was very poor, with the tool unable to synthesise any of test cases. This was the expected outcome, since there is a lack of information available within the type of each function to cut out many of the invalid results, in this case the Nil constructor, which the system consistently returned.

With the added information available with vectors the tool was able to perform better, as seen in Table 2, it was able to correctly synthesise 4 of the test cases. Closer inspection of the results displayed that tools handling of constraints prevented it from synthesising certain definitions, for example, when attempting to synthesise the recursive case for the drop function, `drop a (S n) (S m) (Cons a n x xs)` the tool would reject the correct answer, `drop a n m xs`, as it contained no guarantee that $(S n) - (S m) = n - m$. This result suggests that when synthesising recursive definitions, the tool should take into account the inductive hypothesis when accepting or rejecting terms, as opposed to relying entirely on the built in type system.

When more type information was required, the algorithm performed worse, unable to synthesise terms that previously it was able to. In certain cases it's clear that when provided with the possibility of constructing a base case term, this will be the favoured result. This partially results from the ordering heuristic, which simply selects the first result generated, however it is clear that the system was also unable to deal with the added complexity of generating proofs of correctness.

The Equality set was designed to test the systems ability to use information gained from case splitting, as the only valid right hand side terms rely on it. The tools failure to synthesise the majority of these terms identifies that the tool is not able to effectively generate and utilise type information generated from case splits. The issue with generating new information may lay with the less sophisticated type system found within TinyIdris, this issue presents itself during the unification of meta-variables, where the system is unable to generate constraints. The tool however, makes no attempt to solve constraints manually, therefore it may miss information that it does have available to it.

Another major drawback of the tool displayed by the tests is efficiency. The approach of enumerating the context runs in exponential time, meaning as the number of functions defined in any given context increases, the time of execution is dramatically affected. The process, upon failed attempts, repeatedly ran out of resources when used with a large context. When synthesis is successful, the time required varies drastically, taking under 2 seconds and over 30 for the same function definition.

6.6 Comparison to other systems

With the exception of the Equality test set, the tool performed somewhat comparably to both Idris and Agsy, with the notable difference in efficiency. The existing tools were able to synthesise definitions in under 1 second each. However, the tool was outperformed in each section by the Idris2 system, given the similarities within the systems this suggests that the ordering heuristic used in the Idris2 system is superior, and that recursive functions definitions should be prioritised over attempting each function from the context.

The comparison to Synquid presents some interesting conclusions. The overwhelming performance difference in the Vectors section displays the power of using refinement types and an SMT solver within synthesis applications. Many of the examples used to test the Synquid system by the creators are extremely difficult to express using dependent types, resulting in extremely large and inefficient definitions, the reverse is also true as many of the tests are not expressible only using refinement types. This suggests that some combination of the approaches may lead to a more powerful tool.

6.7 Changing Heuristic

The greater performance of the Idris2 system suggests the heuristic used could provide better results. An initial attempt was made, using the heuristic present in the Idris2 system, which selects the term that uses the the most arguments passed in on the left hand side, not counting arguments that are used multiple times. Since the Idris2 system does not attempt synthesis using functions, instead focusing on recursive calls, testing was also carried out prioritising recursive calls, and only sorting functions, while leaving constructors and local variables in the order they are defined.

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

Table 5: Comparing heuristics

As table 6 shows the inclusion of heuristics was detrimental to the performance of the tool. The exclusion of ordering local variables and data constructors from the final test suggests this is due to the correct answers being mostly data constructors, and therefore prioritising them is beneficial, since the tool is unable to synthesise other definitions which do not rely on this pattern. The poor performance of the differing heuristics suggests the performance issue requires a more sophisticated tool than is currently implemented, allowing the tool to synthesise a greater variety of definitions, which may then be ordered based on updated knowledge of the performance.

6.8 Conclusions

When compared to recent systems, it is clear that the tool suffers as it is unable to effectively utilise type information to refine the search space, which results in slow performance and potentially correct definitions being discounted as not enough information is available. It is clear that the tool requires improvement when synthesising arguments for definitions by refining terms based on new information generated with each successive argument, by generating and utilising constraints. It is clear that using any function definition is an inefficient way to approach the problem, instead it could be beneficial to use functionality such as case expressions and the with abstraction that are not present in the TinyIdris language. This may increase the number of synthesisable terms, without dramatically increasing the search space, for example, allowing functions to perform a recursive call, and operate in a certain way based on the result. The power displayed by Synquid’s type system suggests that an extension of the type system to allow refinements that may be passed to a satisfiability solver could also greatly increase the potential for synthesis without placing a large burden on the programmer to carefully design data types.

References

- [1] Changes since Idris 1 — Idris2 0.0 documentation, Nov 2020. [Online; accessed 22. Nov. 2020].
- [2] Ltac — Coq 8.13.1 documentation, Mar 2021. [Online; accessed 14. Mar. 2021].
- [3] Edwin Brady. The implementation of idris 2, Aug 2020. [Online; accessed 22. Aug. 2020].
- [4] Antonius J. C. Hurkens. A simplification of girard’s paradox. In Mariangiola Dezani-Ciancaglini and Gordon Plotkin, editors, *Typed Lambda Calculi and Applications*, pages 266–278, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.
- [5] Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. Synthesis modulo recursive functions. *SIGPLAN Not.*, 48(10):407–426, October 2013.
- [6] Tristan Knoth, Di Wang, Nadia Polikarpova, and Jan Hoffmann. Resource-guided program synthesis. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 253–268, New York, NY, USA, 2019. Association for Computing Machinery.
- [7] Fredrik Lindblad and Marcin Benke. A tool for automated theorem proving in agda. In Jean-Christophe Filliâtre, Christine Paulin-Mohring, and Benjamin Werner, editors, *Types for Proofs and Programs*, pages 154–169, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.
- [8] Rob Nederpelt and Professor Herman Geuvers. *Type Theory and Formal Proof: An Introduction*. Cambridge University Press, USA, 1st edition, 2014.
- [9] Liam O’Connor. Applications of applicative proof search. In *Proceedings of the 1st International Workshop on Type-Driven Development*, TyDe 2016, page 43–55, New York, NY, USA, 2016. Association for Computing Machinery.
- [10] Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. *SIGPLAN Not.*, 50(6):619–630, June 2015.
- [11] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. *SIGPLAN Not.*, 51(6):522–538, June 2016.

7 Appendix A - User Guide and Maintenance

The system is built using the Idris2 programming language, due to a lack of backwards comparability the required version is 0.2.1 which can be located from the Idris2 GitHub, <https://github.com/idris-lang/Idris2/tree/compat-0.2.1>.

The installation of idris2 requires either chez scheme or racket, details of how to install this can be found at <https://www.scheme.com/>.

To install Idris2 the commands `make bootstrap SCHEME=chez` then `make install` are run. Depending on the version of scheme and operating system the first command may change to `SCHEME=scheme` or `SCHEME=chezscheme`.

The system can be accessed from Github <https://github.com/Ablach/tiny-idris-program-synthesis>. The tool is built using the command `idris2 -install tinyidris.ipkg` while in the TinyIdris directory.

TinyIdris can be run using calling the `tinyidris` executable from the TinyIdris directory using command `./build/exec/tinyidris`, and passing the name of a `.tidr` test file. Test files are located within the `TinyIdris/Test/TestFiles` directory, since the paths are relative the executable must be called from the TinyIdris directory. If a new TinyIdris source file is created, then it should be stored within the `TinyIdris/Test/TestFiles` directory, and an answer file of the same name created in the `TinyIdris/Test/AnswerFiles` directory using the extension `.ans`. The answer file may be left blank.

While in the TinyIdris repl tree 4 commands may be issued. To evaluate and type check an expression type the expression, with nothing before it. To synthesise a definition or term, use the command `auto`, followed by the name of the hole or the function. To test synthesis of an individual term run the command `t`, with the hole name as an argument, or to batch test a group of holes within a file, simply run `test`, with no arguments.

In order for the last two commands to work the answer file must contain solutions for the problem, any holes that do not have a solution will be skipped. Solutions are written in the answer file as `<name> ! <solution>`.

To exit the `tinyidris` repl press `Ctrl-C`.