# Embedded Program Search in Agda

JACK MARTIN and MICHAEL DRESSER, University of Colorado Boulder

The problem of Program Search - looking for a function which satisfies a query - is one of the more practical tools a developer can use when programming. Search tools such as Hoogle provide an entrypoint for a language eco-system, but unfortunately often don't accept query information other than types. Here we present a search tool for searching over types and examples as well as a general framework for doing program search over dependent types. We also provide a type-theoretic interpretation of the program search problem as a special case of the more general synthesis problem and discuss how this presents a tool for furthering the solution to both problems.

**ACM Reference Format:**
Jack Martin and Michael Dresser. 2020. Embedded Program Search in Agda. CSCI 5535, Spring 2020 (May 2020), 8 pages.

## 1 INTRODUCTION

Some of the more powerful tools in use by functional programmers are engines for searching over code bases given the type signature of the desired function. Such tools provide the entry-point for practical code reuse and management of large code bases. Tools such as Hoogle(for Haskell)[5] provide a best-in-class search functionality to polymorphic functional languages, but the same tools are not as prevalent for dependently-typed languages.

There are a number of factors that have prevented robust search features from entering the dependently-typed landscape. The primary setback is the undecidability of type equality(isomorphism) in the presence of dependent types. For search over types to be performed there must be some way of saying whether two types are equal or not. This introduces many of the nuances of equality into the confusion of trying to build a practical tool.

We attempt to solve this problem by defining an abstraction for search engines of dependent types that can scale and adapt to appropriate definitions of type equality. It is reasonable for a user to want some control over how weak/strong the equalities for search are. Some may want strong definitional equalities, others may accept the more computationally burdensome(and semi-decidable) isomorphisms. We give an approach that leaves this control to the user, and also leaves open the possibility for very powerful forms of program search.

Another deficiency of current search engines for program source is the lack of being able to specify complex functional properties over the domain of search. Input-output examples and algebraic properties offer two great sources of specification for search, yet they are not used; typically this is because of the lack of methods for verifying these properties. Fortunately, a dependently typed language offers a natural way of handling these properties.

Practically, we provide all of the abstractions formalized in Agda[6] presented as a concrete tool for search over terms with some basic types. Notably, our method is embedded *within* the type theory such that the search function can be treated just like any other function. The implementation is capable of being extended to support more expressive properties which in turn will provide

Authors' address: Jack Martin, John.P.Martin@colorado.edu; Michael Dresser, Michael.M.Dresser@colorado.edu, University of Colorado Boulder.

more power to a user to specify complex queries. In the future, the theories in this paper will be developed into a search-directed approach to program synthesis.

## 2 OVERVIEW

### 2.1 Program Search

An advantage of the functional programming paradigm is the ability to compose programs from simpler functions. Large libraries exist containing many of the simple(and not-so-simple) functions that a programmer would like to avoid writing. However, the user of a functional language often runs into the situation where they have an idea of the function they need, but don't know the name or even where to begin looking. A robust tool for searching over the available libraries is the obvious solution.

Let's assume we would like to search for a program that operates on two natural numbers. We have `plus` in mind, but can't remember the name of the function nor the module it may be located in. But we do know its behavior and type. The type of the function is $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$. We also can describe its behavior over a few input-output examples:

| input 1 | input 2 | output |
|---|---|---|
| 0 | 0 | 0 |
| 2 | 2 | 4 |
| 1 | 2 | 3 |
| … | … | … |

The benefit of working in a dependent type theory is that we can encode these examples directly in our query type. So instead of $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$, we have the *(iterated) dependent sum*:

$$\begin{aligned}\Sigma[f \in \mathbb{N} \to \mathbb{N} \to \mathbb{N}]\ (f\ 0\ 0 \equiv 0) \\ \times (f\ 2\ 2 \equiv 4) \\ \times (f\ 1\ 2 \equiv 3) \\ \times \ldots\end{aligned}$$

This type corresponds to the function $f$ of type $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$ along with proofs that the function satisfies our examples. A term of this type with `plus` as the function $f$ could look like:

$$(\texttt{plus}\ ,\ \texttt{refl}\ ,\ \texttt{refl}\ ,\ \texttt{refl}\ ,\ \ldots)$$

Where `refl` is a reflexivity proof that a given example holds. The search for this term is quite simple. First, search for a term matching the type of the query, then check if the examples are satisfied by a reflexivity proof. Finding a term of the query type is fairly straightforward, but to find a proof that each example is held requires a bit more. Notably, the type of the output has a decidable (judgemental) equality. This is a property that would typically be proven in the same module as the types definition. Because of this, we can't make any assumptions about a given type possessing such a decidable equality. We should make the search procedure general enough that it can search for that decision procedure.

This example is given to point out that the search methods we aim to define must be able to interact, to some degree, with the modules they search over. Simply looking for a matching type is not enough. For these reasons our search methods are embedded *within* the type system as opposed to externally in the form of a tool or executable.

# 3 PRELIMINARIES

## 3.1 Dependently Typed Programming in Agda

This subsection briefly introduces the necessary ideas from dependently-typed programming necessary to understand the rest of the paper. Our paper will use a notation that roughly corresponds to that of the theorem prover Agda. We use datatype declaration of the form:

$$\begin{aligned}&\text{data } T : \tau_1 \to \tau_2 \to \ldots \to \text{Set where}\\ &\quad \mathsf{C}_1 \quad : T\ e_{1,1}\ e_{1,2}\ \ldots\\ &\quad \mathsf{C}_1 \quad : \mathbb{N} \to T\ e_{2,1}\ e_{2,2}\ \ldots\\ &\quad \ldots\\ &\quad \mathsf{C}_n \quad : T\ e_{n,1}\ e_{n,2}\ \ldots\end{aligned}$$

Where $T$ is the name of the type(family) being declared. The types $\tau_i$ represent the indices for a type family. Set is the type of types. $\mathsf{C}_i$ are the constructors for the newly declared type and the right-hand side of the constructor definition is the type of the constructor along with the specific type in the family(specified with terms $e_{i,j}$ from the indice-types) to which it belongs.

We specify iterated sums as record types in the style of Agda. These can be thought of as representing mathematical structures that are composed of several elements of different(potentially dependent) types. An example is the semigroup record:

$$\begin{aligned}&\text{record IsSemigroup } (A : \text{Set}) : \text{Set where}\\ &\quad \_\cdot\_ \quad : A \to A \to A\\ &\quad \text{assoc} \quad : \text{Associative } (\_\cdot\_)\end{aligned}$$

This is parameterized over a type $A$ and defines a type containing a binary operation $\_\cdot\_$ along with a proof that $\_\cdot\_$ is associative. Note that the term assoc is dependent on the term $\_\cdot\_$. The underscores in the semigroup operation denote positions for arguments to the operation. Agda is unique from most languages in that it supports this style of mixfix operators. As an example, applying $\_\cdot\_$ to arguments $a$ and $b$ would be $a \cdot b$.

## 3.2 Universes

One of the advantages of dependent type theories is their ability to formalize their own type system and meta-theories. These techniques typically use a structure called a *Universe*, which is defined as a type along with an interpretation function which takes an element of that type and computes some type in the type system. Formally:

$$\begin{aligned}&\text{record } \mathcal{U} : \text{Set where}\\ &\quad \text{Carrier} \quad : \text{Set}\\ &\quad [\![\_]\!] \quad : \text{Carrier} \to \text{Set}\end{aligned}$$

Here, Carrier is a type containing codes. These codes correspond to types in Agda's type system and $[\![\_]\!]$ is the interpretation function, which takes an element of Carrier and computes its corresponding Agda-type. An example of such a universe containing booleans and natural numbers is given here:

$$\begin{aligned}&\text{data Type : Set where} \qquad [\![\_]\!]_{\text{Type}} : \text{Type} \to \text{Set}\\ &\quad \text{Number : Type} \qquad [\![\text{Number}]\!]_{\text{Type}} = \mathbb{N}\\ &\quad \text{Boolean : Type} \qquad [\![\text{Boolean}]\!]_{\text{Type}} = \mathbb{B}\end{aligned}$$

We will spend most of the rest of the paper defining a suitable Carrier type for simulating the type system enough for type-based search.

# 4 EMBEDDING SEARCH

## 4.1 Simulating First Class Modules

We will need a coarse simulation of Agda's module structure to store entries for search. This will be accomplished by collecting elements of a Carrier type along with the terms they encode into a homogeneous list. The first element will be the coded type, and the second element(which depends on the first), will be the term of the type that is interpreted from the code:

$$\begin{aligned}&\text{Module} : \text{Set}\\&\text{Module} = \text{List}\ (\Sigma[\tau \in \text{Carrier}]\ [\![\tau]\!])\end{aligned}$$

We can construct an example of a module using our example universe from the previous section:

$$\begin{aligned}\text{example} = \quad &(\text{Boolean}\ ,\ \text{true})\\ :: \quad &(\text{Number}\ ,\ 5)\\ :: \quad &(\text{Boolean}\ ,\ \text{false})\\ :: \quad &[]\end{aligned}$$

This new data structure will allow for us to work with terms along with their types as first class entities as long as we can encode them into this module structure.

## 4.2 Search over First Class Modules

To support basic type search over these modules we will first have to discuss what it means for a query type to "match" a type in a collection. This clearly introduces the need for a notion of equality over types, which complicates matters quickly. Which form of equality of types is appropriate for use in program search? Fortunately, there has been some research done on this problem and the current accepted equality is isomorphism of types[4]. This will equate curried function types to their uncurried counterparts. For example(i.e. $(A \times B) \to C \approx A \to B \to C$).

A problem with using isomorphism over dependent types is its undecidability[7]. The field is currently studying certain decidable subsets of the problem thus we opt to maximize flexibility by parameterizing our methods over a provided decidable type equality. The problem of type-directed program search then just becomes the act of running the decision procedure for this equality over the coded-types in our module. Thus, a searchable universe can be defined:

$$\begin{aligned}&\text{record}\ \mathcal{U}_S : \text{Set where}\\&\quad U \qquad\qquad\quad : \mathcal{U}\\&\quad \_ \approx \_ \qquad\quad : (a\ b\ : \text{Carrier}\ U) \to \text{Set}\\&\quad \approx\text{-isEquiv} \quad : \text{IsEquivalence}\ (\text{Carrier}\ U)\ \_ \approx \_\\&\quad \_ \overset{?}{\approx} \_ \qquad : \forall(a\ b : \text{Carrier}\ U) \to \text{Dec}\ (a \approx b)\end{aligned}$$

Where IsEquivalence is a proposition that $\_ \approx \_$ is an equivalence over the Carrier type and $\mathcal{U}$ is the universe record-type defined above. Now we are ready to define an abstract search procedure for modules of a decidable universe. The type of the search function is:

$$\text{search} : (m : \text{Module}) \to (c : \text{Carrier}) \to \text{List}\ [\![c]\!]$$

The function takes in a module of the universe $m$, and a coded-type $c$. The output type of the function is a list of inhabitants of the interpretation of $c$. In the remainder of the paper we will construct a concrete Universe with which we will implement a small embedded search procedure.

```
data TypeCode : ℕ → Set where
  U       : ∀{n} → TypeCode n
  #       : ∀{n m} → n < m → TypeCode m
  _ ⇒ _   : ∀{n} (t₁ t₂ : TypeCode n) → TypeCode n
```

Fig. 1. The Type Codes for our Universe supporting Search

## 5 SEARCHING OVER A SUBSET OF AGDA

To implement the formalism given in the previous section and create a practical search procedure, we begin by defining the universe of typing-codes corresponding to the Carrier field of the record type for universes. This definition is given in Figure 1. $U$ is the code for terms of type Set. # is the reference type which is used to refer to a type that is defined elsewhere in the module, an example of its use will be shown below. _ ⇒ _ is the code for function types. The natural number that indexes TypeCode is used for book-keeping purposes to ensure we don't ever over-index a module or create any circular references. It is a technical detail that we will not elaborate on.

Modules need to be redefined(See Figure 2) from the abstraction in the previous section because they allow for codes to reference other definitions. We will define Module similarly to the type synonym we used above, except that it will be mutually defined with the interpretation function. Note that for the _ :: _ case we provide a module first in the constructor so that the entry may depend on it. The entries in the module will be similar to that in the example module in Section 4. In the same way the dependent pair did in the previous Module definition.

The interpretation function is given in Figure 3. As explained before, this function is responsible for mapping elements of TypeCode to their respective types in Set. The cases are fairly straightforward except for references. The reference constructor # contains a proof that the index does not go over the length of the list. This is constrained by the indices mentioned before. The proof includes with it the position of the type we wish to reference and then the interpretation function looks it up via a recursive call.

To make all of this a bit more concrete, Figure 4 is an example of a module encoded into this typing universe. Here each entry is preceded by its type in the coded type system. Note that the references properly index the types of their functions or terms.

## 6 EMPIRICAL EVALUATION - EASE OF USE

This formalism thus far is lacking ergonomics. The tool is rather impractical because the module encoding needs to be written by hand in a coded language that relies heavily on numerical indexing. We alleviate this problem with a set of programs which both generate the coded modules and the

```
data Module : ℕ → Set where
  []        : Module 0
  _ :: _, _ : ∀{n}
            → (Γ : Module n)
            → (t : TypeCode n)
            → ⟦t⟧Γ
            → Module (suc n)
```

Fig. 2. Modules Re-Defined

$$
\begin{aligned}
&[\![\_]\!]\_ : \forall\{n\} \to \text{TypeCode } n \to \text{Module } n \to \text{Set} \\
&[\![U]\!]_\Gamma = \text{Set} \\
&[\![t_1 \Rightarrow t_2]\!]_\Gamma = [\![t_1]\!]_\Gamma \to [\![t_2]\!]_\Gamma \\
&[\![\#\ \text{suc-leq-suc ( zero-leq-n)}]\!]_{(\Gamma :: U, \text{term})} = \text{term} \\
&[\![\#\ \text{suc-leq-suc ( zero-leq-n)}]\!]_{(\Gamma :: t, \text{term})} = \langle \text{Error Case - Impossible under assumptions} \rangle \\
&[\![\#\ \text{suc-leq-suc ( suc-leq-suc } n)]\!]_{(\Gamma :: t, \text{term})} = [\![\#\ \text{suc-leq-suc } n]\!]_\Gamma
\end{aligned}
$$

Fig. 3. The Interpretation Function

```
ex : Module _
ex = [] :: U              , Nat
        :: ⟨ 1 ⟩          , 5
        :: ⟨ 1 ⟩          , 2
        :: U              , Bool
        :: ⟨ 4 ⟩          , true
        :: ⟨ 4 ⟩          , false
        :: ⟨ 4 ⟩ ⇒ ⟨ 4 ⟩ , not
```

Fig. 4. A Sample Module. The ⟨_⟩ used in this example corresponds to a shorthand for generating the safety proof automatically instead of writing it out long-hand with #. These can be read as references where the number inside the brackets is the location of the type to which it refers.

coded queries in TypeCode. These two programs hook into the Agda Compiler to parse the target library files and generate an auxiliary file containing the coded modules and queries. The queries will need to be provided as codes in TypeCode. The user provides the syntax valid for Agda and the tool generates the element of TypeCode corresponding to the type provided by converting its base types to the proper references and function arrows to the coded-arrows.

Along with these programs we have created an integration with emacs to run the indexing program whenever emacs saves an Agda file. The query executable can be called using a shortcut or from the list of available emacs commands. We have found that the inclusion of these features made the search procedure practical for the end-user.

## 7 RELATED WORK

*Program Search.* Program Search, or the search of functions in some library, has led to such tools as Hoogle, and is the main category that this work could be included under. In the past, none of these tools use any functional properties or examples to help guide the search. Ours is the first tool to provide this functionality in a general framework. It is also the first, to our knowledge, to formalize the task of type-directed search using decidable properties of types.

*Typing Universes.* There has been much work on universes for generic programming. Benke et al. create a collection of category-inspired universes specifically with the aim of formalizing past work on generic programming in the vein of Generic Haskell [2]. This work differs from ours in its aim, it mainly attempts to automate the writing of functions such as generic map or fold which have a common behavior no matter the structure of the type. We attempt to be generic over the types themselves but with a more concrete aim. This makes this approach to abstract for our use. Another line of work aims at formalizing type theory in type theory using quotient inductive types[1]. This work is closer to our own in that they encode their universe mutually in a

similar way to our encoding. However this work aims at verifying type theory within itself and makes heavy use of Homotopy Type Theory[7]. The formalism itself is not quite adequate for our purposes as it lacks an mechanism for interpreting its type encodings.

## 8 FUTURE WORK

### 8.1 Ergonomic Improvements

Our toolchain for generating a module from Agda files will always be apt for improvement. Ideally, there will be a full integration with the Agda compiler's interactive mode. This will speed up the generation program as well as eliminate the need for separate executables.

Supporting the generation of first class modules from multiple files is also a near term improvement that is technically not too difficult. The name spaces will need to be qualified and import conflicts resolved. This will open up the practical use of our tool for searching over a large repository of code such as the Agda Standard Library.[3]

### 8.2 Universe for Fully Dependent Types

One of the items lacking in this project was a successful encoding of a universe for fully dependent types. There was a prototype produced that had some deficiencies preventing it from having a sound interpretation function. This will be the first follow-up work to be conducted. Completing this universe will let the search procedure handle example-based searches such as was show in the overview section. This need not stop at examples however. Any decidable property over terms/functions can be used in the search process. Thus this formalism will be much stronger than any program search tools currently available for functional languages.

### 8.3 Program Search as Synthesis

A key realization motivating this line of research is the relation of program search to the more general problem of program synthesis. Program Synthesis is the task of generating a program which satisfies some specification. Types provide one such specification. If the synthesis task is then reduced to producing some term of a given query type, the synthesis procedure must produce an abstract syntax tree that typechecks to the query type. If we restrict the solution to have an AST of size 1 it essentially forces the synthesis problem to become program search.

This is not a particularly profound idea. However, when considering program search as foundational operation in synthesis procedures it becomes more motivating. This was the reasoning behind formalizing search within Agda as opposed to as a separate tool. With a function we can compose search with other functions to build a complex framework for synthesis over dependent types. This is our main line of future work in this area.

## 9 CONCLUSION

We have developed a formalism for and implemented the foundations of a search tool over Agda codebases. This tool allows Agda programmers to search modules for terms that match a user-specified type. It can be readily expanded to handle Agda's full type system. Following this expansion, a series of work will be opened up that will both enable Agda programmers and eventually lead to development of dependently-typed synthesis.

## REFERENCES


[1] Thorsten Altenkirch and Ambrus Kaposi. 2016. Type theory in type theory using quotient inductive types. *ACM SIGPLAN Notices* 51, 1 (2016), 18–29.

[2] Marcin Benke, Peter Dybjer, and Patrik Jansson. 2003. Universes for generic programs and proofs in dependent type theory. *Nord. J. Comput.* 10, 4 (2003), 265–289.

[3] Ana Bove, Peter Dybjer, and Ulf Norell. 2009. A brief overview of Agda–a functional language with dependent types. In *International Conference on Theorem Proving in Higher Order Logics*. Springer, 73–78.

[4] Roberto DiCosmo. 2012. *Isomorphisms of Types: from?-calculus to information retrieval and language design.* Springer Science & Business Media.

[5] Neil Mitchell. 2008. Hoogle overview. *The Monad. Reader* 12 (2008), 27–35.

[6] Ulf Norell. [n. d.]. *Towards a practical programming language based on dependent type theory*. Vol. 32. Citeseer.

[7] The Univalent Foundations Program. 2013. Homotopy Type Theory: Univalent Foundations of Mathematics. *arXiv preprint arXiv:1308.0729* (2013).