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 (April 2020), 5 pages.

1 INTRODUCTION

One 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) 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 which defines a search engine for 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 because of the lack of some method 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 and presented as concrete tool for search over functions with input-output examples. The implementation is capable of being extended to support more expressive properties which in turn will provide more power to a user to specify complex queries.

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

2 OVERVIEW

Let's start with an example of a search query over a non-dependent type, given as a type:

$$\text{query} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$$

Solutions for this query would obviously include the usual arithmetic operations. But let's say that we wanted to further specify the query by providing some input-output examples:

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

We could devise the obvious procedure of checking the resulting functions against these examples and discarding those that are not consistent. In fact, this is exactly what we will do. But to jump to the algorithm would fail to realize the deeper type-theoretic possibilities here. See, the examples above can be encoded quite easily as a type for our query. The type below is consistent with the query:

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

This is a *dependent product* type that encodes our query as searching for a term that satisfies the type we gave along with proofs that the provided function produces certain results under evaluation.

3 PRELIMINARIES

Note for Benno and Evan: The citations are absent for now. We will add these for the final paper but decided to leave them out for now until the structure of the paper is fixed. The notes for "Needs citation" indicate positions where we have a reference saved somewhere. It is not an indication that we need to still find it.

3.1 Universes

One of the advantages of dependent type theories is their ability to formalize their own type system and meta-theories within themselves(**Needs citation**). 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:

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

An example of such a universe containing booleans, natural numbers, and functions on either is given here:

$$\begin{array}{lll} \text{data Type} : \text{Set where} & & [\![\_]\!]_{\text{Type}} : \text{Type} \to \text{Set} \\ \quad \text{Number} : & \text{Type} & [\![\text{Number}]\!]_{\text{Type}} = \mathbb{N} \\ \quad \text{Boolean} : & \text{Type} & [\![\text{Boolean}]\!]_{\text{Type}} = \mathbb{B} \\ \quad \_ \Rightarrow \_ : & \text{Type} \to \text{Type} \to \text{Type} & [\![t_1 \Rightarrow t_2]\!]_{\text{Type}} = [\![t_1]\!] \to [\![t_2]\!] \end{array}$$

Soon, we will define a universe of types for use in program search. We will spend most of the rest of the paper defining a suitable Carrier type that respects the various complexities that arise from the presence of dependent types.

3.2 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:

$$\mathsf{Module} = \Sigma[\tau \in \mathsf{Carrier}][\![\tau]\!]$$

Carrying on with our example universe above, an example module containing some common functions and elements:

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

4 SEARCHING OVER UNIVERSE 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(**Needs Citation**). This will equate curried function types to their uncurried counterparts for example(i.e. $(A \times B) \longrightarrow C \approx A \longrightarrow B \longrightarrow C$).

A problem with using isomorphism over dependent types is that it is a generally undecidable problem(**Needs Citation**). 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:

define isEquiv

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

Now we are ready to define an abstract search procedure for modules of a decidable universe. The type of the search function is:

$$\mathsf{search} : (u : \mathcal{U}_S) \longrightarrow (m : \mathsf{Module}_u) \longrightarrow (c : \mathsf{Carrier}\ (U\ u)) \longrightarrow \mathsf{Maybe}\ [\![c]\!]$$

The procedure takes in a decidable universe u, a module of that universe m, and a coded-type c. The output type of the function is possibly an inhabitant of the interpretation of c provided that there is in fact one in the module.

5 A UNIVERSE FOR SEARCHING OVER DEPENDENT TYPES

Thus far, the formalism we have given should allow any non-dependent program to be searched. To add support for programs with type dependencies we will need to slightly alter our definition of Module. The main idea is that the module will need to be simultaneously defined with the

$$
\begin{array}{lll}
\text{data Module} : \mathbb{N} \longrightarrow \text{Set where} & & \\
\quad [\,] & : \text{Module } 0 & \\
\quad \_ :: \_ , \_ & : \forall \{n\} & \\
 & \longrightarrow (\Gamma : \text{Module } n) & \\
 & \longrightarrow (t : \text{TypeCode } n) & \\
 & \longrightarrow [\![t]\!]_\Gamma & \\
 & \longrightarrow \text{Module } (\text{suc } n) &
\end{array}
\qquad
\begin{array}{ll}
\text{data TypeCode} : \mathbb{N} \longrightarrow \text{Set where} & \\
\quad U & : \forall \{n\} \longrightarrow \text{TypeCode } n \\
\quad \# & : \forall \{n\ m\} \longrightarrow n < m \longrightarrow \text{TypeCode } m \\
\quad \_ \Rightarrow \_ & : \forall \{n\}\ (t_1\ t_2 : \text{TypeCode } n) \longrightarrow \text{TypeCode } n
\end{array}
$$

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

Fig. 1. The Universe for Search over Dependent Types

interpretation function so that types defined in a "cons" of a module can depend on the terms found in earlier entries.

The definition of this universe is given in Figure 1.

Note for Benno and Evan: There is not any explanation or elaboration for this part of the formalism yet. There are also a few missing constructors. We were stuck on this for the better part of the last week and just figured out how to formalize it last night and it isn't quite finished. In the final paper this will be much, much more spelled out over 3-5 pages.

6 EMPIRICAL EVALUATION

We were unsure what to include in this section because most of our project was correct by construction. Benno and Evan, we will touch base with you about your thoughts as to what to include here. We brainstormed a little bit but couldn't settle on anything that made too much sense. Hoping that the sections above will be suitable for this week's version of the draft. One idea we had would be to show off an example of why the embedded program search is so powerful/useful. This seems to be a somewhat empirical claim that we could spend some time fleshing out.

7 RELATED WORK

Edit: Make changes as Benno suggested

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.

draft note: make sure to look into the search programs for other languages.

Program Synthesis. As was mentioned in Section # above, program search can be viewed as a special case of the more general program synthesis problem. It seems hopeful that program search can be later used as a tool in program synthesis by treating it as an atomic operation for larger synthesis procedures.

Type Isomorphism. Isomorphisms over types acts as the equivalence relation that we use for search in our approach. In reality our approach defines a slightly more robust isomorphism relation over types *and* examples. This is slightly more powerful than just the types themselves. In future work we will look at the theoretical basis for this.

8 CONCLUSION

ACKNOWLEDGMENTS

TBD

REFERENCES