Applications of PPCA

RnD Report
Submitted in partial fulfilment of requirements for the degree of

Bachelor of Technology (Honours)

by

Aditya Kumar Akash
Roll No : 120050046

under the guidance of
Prof. Suyash Awate

Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Mumbai 400076, India

November, 2015

Contents

Abstract

Acknowledgements

I am sincerely indebted to my advisor, Prof. Suyash Awate, IIT Bombay, for his constant support and guidance throughout the course of this project. His experience and insight in the fields of machine learning, image processing and varied aspects of computer science in general, was valuable in boosting my interest on this topic.

I finally and especially would like to thank my parents, my brother and my whole family for their support and trust in all of my endeavours. The morals they have imparted, stayed and would stay close to me always. I also thank my friends for giving a helping hand during hard times.

Chapter 1

Introduction

Algorithms are the computational content of Proofs.

Robert Harper, Benjamin C. Pierce
Software Foundations

With more and more tasks in our day to day life getting automated through software, we have felt a convenience like never before. That software has helped reduce human burden in terms of computation is a trivial observation. However, on deeper inspection, one can see that computer programming has offered a useful abstraction viz. letting humans focus only on developing algorithms for computation and leave out the actual computations inessential to this process, for the computer. In essence, it has decoupled the issues of creative thinking and efficiency. There is no reason why you cannot take this abstraction to a higher level, that of automating meta-computation i.e. the process of generating computer programs. This is the central issue of *Program Synthesis*. Program synthesis [1] is the task of automatically discovering an executable piece of code given user intent expressed using various forms of constraints such as input-output examples, demonstrations, natural language, etc.

1.1 Motivation

Some traditional issues [11] we have been facing for decades in the field of software production are the cost of non-standard software, due to long development times and the constant need for maintenance, and a lack of confidence in the reliability of software. Accidents like the crash of KAL's 747 in August 1997, the Therac-25 medical radiation therapy device's over-dosage error in 2000, etc are some examples of software errors causing serious repercussions. Program synthesis offers a solution to the above issues by automating much of the mundane parts of code generation and also establishing correctness of these programs through *Formal Verification* techniques. Though these

[1] adopted partly from http://research.microsoft.com/en-us/um/people/sumitg/pubs/synthesis.html

issues persist to be the main driving force of program synthesis, the spectrum of benefits it offers has broadened over time. Here are some ways in which program synthesis holds direct applications for various classes of users in the technology pyramid :

- (100s of millions of) End Users (people who have access to a computational device but are not expert programmers): Helping them to create small snippets of code for performing repetitive tasks, simple data manipulation. In other words, enabling them to bring their creativity to life!
- (Billions of) Students and Teachers: Intelligent tutoring systems that support solution generation (the step-by-step solution to a problem is like a program! PLDI 2011[8], IJCAI 2013b[1]), problem generation (of a certain difficulty level and that exercises use of certain concepts AAAI 2014[2], IJCAI 2013b[1]), automated grading (PLDI 2013 [14]) , and digital content creation (CHI 2012 [3]). Interestingly, all of these activities can be phrased as program synthesis problems.
- Software Developers: Help synthesize mundane pieces of code.
- Algorithm Designers: Help discover new algorithms.

1.2 Outline

We move ahead with the discussion on program synthesis in chapter 2, where we would be illustrating it using examples. Then we briefly see various dimensions along which program synthesis can be performed. Later in the chapter we explain why functional programming is a lucrative platform for program synthesis and the benefits it holds in comparison with imperative programming. In chapter 3, we study the features of Coq, an interactive theorem prover based on functional programming, that make it particularly useful for program synthesis. In chapter 4, we discuss about the problem of higher order unification, and a restricted yet powerful version of the unification problem that Ankit has implemented in Coq. Later in chapter 5, we study the problem of finding the "longest prefix of a list satisfying p" problem. There we go through parts of a previous derivation by Ankit, for solving the problem in linear time. We suggest a way involving input-output examples that helps assist/automate parts of the derivation which would otherwise require human insights. We then briefly discuss the benefits of the proposed approach over $Igor2$, a pure inductive synthesis algorithm. In chapter 6, we conclude by summarizing the entire work and laying out future goals.

Chapter 2

Background

As we have seen, Program Synthesis is the task of discovering an executable piece of code from user intent given a high level specification of the goal. As we shall see later, this high level specification can take different forms and in fact this becomes a dimension [7] along which program synthesis can be classified. For now, let us understand program synthesis through an example.

2.1 An Example of Program Synthesis

Below is a toy example [1], of derivation of a functional program to compute the maximum M of two numbers x and y.

No.	Assertions	Origin
1	$A = A$	Axiom
2	$A \leq A$	Axiom
3	$A \leq B \vee B \leq A$	Axiom

Table 2.1: Some basic axioms related to comparison

To begin with, we consider the assertions mentioned in table 2.1 as our axioms. As you can see, these axioms are very basic and just state the laws of trichotomy. However, we make use of these to derive a functional program that computes the maximum of two numbers; something that is non-trivial to obtain from just these axioms (at least for a computer).

Starting from the requirement description "The maximum is larger than any given number, and is one of the given numbers", the first-order formula $\forall X \forall Y \exists M : X \leq M \wedge Y \leq M \wedge (X = M \vee Y = M)$ is obtained as its formal translation. This formula is to be proved. By reverse Skolemization, the specification in line 10 (in table 2.2) is obtained,

[1]adopted from `https://en.wikipedia.org/wiki/Program_synthesis`

No.	Goal	Program	Origin
10	$x \le M \wedge y \le M \wedge (x = M \vee y = M)$	M	Specification
11	$(x \le M \wedge y \le M \wedge x = M) \vee (x \le M \wedge y \le M \wedge y = M)$	M	Distr(10)
12	$x \le M \wedge y \le M \wedge x = M$	M	Split(11)
13	$x \le M \wedge y \le M \wedge y = M$	M	Split(11)
14	$x \le x \wedge y \le x$	x	Resolve(12,1)
15	$y \le x$	x	Resolve(14,2)
16	$\neg(x \le y)$	x	Resolve(15,3)
17	$x \le y \wedge y \le y$	y	Resolve(13,1)
18	$x \le y$	y	Resolve(17,2)
19	$true$	$x \le y ? y : x$	Resolve(18,16)

Table 2.2: Example synthesis of maximum function using the axioms in table 2.1

an upper- and lower-case letter denoting a variable and a Skolem constant, respectively. After applying the distributive law in line 11, the proof goal is a disjunction, and hence can be split into two cases, viz. lines 12 and 13. Turning to the first case, resolving line 12 with the axiom in line 1 leads to instantiation of the program variable M in line 14. Intuitively, the last conjunct of line 12 prescribes the value that M must take in this case. Formally, this conjunct unifies syntactically with the axiom on line 1 by substituting x for M as well as A. On making this substitution throughout our goal in line 12, we reach the goal in line 14, the first conjunct of which is trivially true through the axiom in line 2. Thus for the case of $\neg(x \le y)$, we obtained the program to be x. A similar reasoning for the goal in line 13 leads us to the program y for the case of $x \le y$. We combine both these goals to obtain a goal in line 19 which is *true* always. The program corresponding to this line is the one we aimed to compute.

2.2 A Different Example of Program Synthesis

In the above example, we have seen the computation of a program starting from a specification that is essentially a logical representation of our goal. It relates logically, the input and output of our program. Though logical representations present a succinct way to provide a formal specification of our desired program, they are often difficult to obtain and use in practice. On the contrary, having an inefficient program to begin with, is often quite useful as we have a correct and working program at the very start. This can continuously be written into a more and more efficient program through calculation rules, such that the correctness in preserved at each step. Both these methods of problem specification are only different ways of conveying user intent. We discuss this in detail in the next section.

Let us now look at the following example of program derivation, where we want to obtain a program for calculating the minimum element among a list of integers. Consider the following functional specification for doing the same.

$$\begin{aligned}
minimum &= head \circ sort \\
sort[] &= [] \\
sort(x : xs) &= insert\, x\, (sort\, xs) \\
insert\, a\, [] &= [a] \\
insert\, a\, xs &= if\ (a < head\, xs)\ then\ (a : xs)\ else\ (head\, xs) : (insert\, a\, (tail\, xs)) \\
head\, (x : xs) &= x
\end{aligned}$$

We derive a recursive definition of minimum from the above specification, by inducting on the input list.

$$\begin{aligned}
minimum\, [x] &= head(sort\, [x]) \\
&= \{\textbf{unfolding sort}\} \\
&\quad head\, (insert\, x\, (sort\, [])) \\
&= \{\textbf{definition of sort}\} \\
&\quad head\, (insert\, x\, []) \\
&= \{\textbf{definition of insert}\} \\
&\quad head\, [x] \\
&= \{\textbf{definition of head}\} \\
&\quad x
\end{aligned}$$

$$\begin{aligned}
minimum\, (x : xs) &= head\, (sort\, (x : xs)) \\
&= \{\textbf{unfolding sort}\} \\
&\quad head\, (insert\, x\, (sort\, xs)) \\
&= \{\textbf{unfolding insert}\} \\
&\quad head\, (if\ (x < head\, (sort\, xs))\ then\ x : (sort\, xs) \\
&\qquad\qquad else\ (head\, (sort\, xs)) : (insert\, x\, (tail\, (sort\, xs)))) \\
&= \{\textbf{if promotion rule : f(if e}_1\textbf{ then e}_2\textbf{ else e}_3\textbf{)} = \\
&\qquad\qquad \textbf{if e}_1\textbf{ then (f e}_2\textbf{) else (f e}_3\textbf{)}\} \\
&\quad if\ (x < head(sort\, xs))\ then\ head(x : (sort\, xs)) \\
&\quad else\ head\, (head\, (sort\, xs) : insert\, x\, (tail\, (sort\, xs)))
\end{aligned}$$

$$= \{\textbf{definition of head}\}$$
$$if\ (x < head(sort\ xs))\ then\ x\ else\ head(sort\ xs)$$

$$= \{\textbf{abstracting out head(sort xs) and folding back minimum}\}$$
$$\begin{aligned} &if\ (x < y)\ then\ x\ else\ y \\ &\quad where\ y = minimum\ xs \end{aligned}$$

2.3 Dimensions of Program Synthesis

We know that compilers and assemblers mostly translate a program written in a structured high-level language into a low-level or machine language in a syntax-directed fashion. While on the other hand, program synthesizers take as input the user intent through a variety of constraints such as logical relations between inputs and outputs, partial or inefficient programs, natural language, input-output examples, etc. Then they process these constraints and come up with an output program after searching through a space of programs. In this section, we briefly describe the dimensions along which program synthesizers may be classified, partly in accordance to Sumit Gulwani [7], excerpts from which have been adopted here.

2.3.1 Input Specification

One of the most important aspects of program synthesis from the user's perspective is the mechanism to specify his intent. As we see below, some of the main choices possible for this are logical specifications, natural language specifications, input-output examples and higher-order, inefficient or partial programs. Depending on the user's skill and the task underlying, a particular choice of the specification may be more suited.

Logical Specifications

A logical specification is a logical relation between inputs and outputs of a program. It can act as a precise and succinct form of functional specification of the desired program. For example, the logical specification for a program that finds out the maximum of two numbers x and y as M is as follows (recall from the example in section 2.1):

$x \leq M \land y \leq M \land (x = M \lor y = M)$

Clearly the above logical expression relates the inputs and the output and has to hold for any program that computes the maximum correctly. It merely states that the maximum must be greater than or equal to both the numbers and should be equal to one of them. As you can see, this is only a specification of our end goal and it says nothing about how we arrive at a program which satisfies it.

As another example, consider any sorting algorithm for a list of numbers. The logical specification for the sorted list S of a given list L of size n would assert that S is a *permutation* of the list L and has its elements *sorted.*

Though synthesis systems that accept user intent in the form of logical specifications exist, compared to other forms of specifications such as input-output examples and partial/inefficient programs, logical relations require additional knowledge of logic. Further they might be harder to get right, and may not be a preferred form of specification for end-users.

Input-Output Examples

In several scenarios, input-output examples can act as the simplest form of specification, with relatively little chances of error. Quite contrary to what it may seem initially, input-output examples, in conjunction with interactive rounds, can often play the role of a full functional specification.

It is natural to ask what prevents the synthesizer from synthesizing a trivial program that simply performs a table lookup as follows, when provided with the set $\{(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)\}$ of input-output pairs.

```
switch x
   case x_1: return y_1;
   case x_2: return y_2;
      .
      .
      .
   case x_n: return y_n;
```

We can defend ourselves from obtaining such trivial solutions by restricting the search space of our function. In particular, we could allow only for a bounded number of statements or conditionals. Another concern with input-output examples is the selection criterion and the number of input-output examples. In other words, identifying what constitutes a good input-output example, and how many examples should the user provide? The user may start by providing few input-output examples (possibly a couple of examples for each of the corner cases) and then add more input-output examples in each interactive round. The interaction may either be driven by the user or by the synthesizer, briefly described below.

User Driven Interaction The user inspects the program returned by the synthesizer, studies/tests it to find any discrepancy in the behaviour of the program and the expected behaviour on a new input. If so, the user repeats the synthesis process after adding the new input-output pair to the list of input-output examples.

Synthesizer Driven Interaction Given a set of input-output pairs, the synthesizer searches for programs that map each input in the given set to the corresponding output.

Assuming that the search space is restricted, there would only be a few satisfying programs. If the synthesizer finds at least two satisfying programs P_1 and P_2, it declares the user specification to be partial. In such a case, it generates a *distinguishing input*, on which both programs generate different outputs, and presents it to the user asking for its corresponding output. The synthesis process is then repeated with this input-output pair added to the previous examples.

This type of synthesis, where search for programs is driven by input-output examples is known as *Inductive Synthesis*. This is the subject of Emmanuel Kitzelmann's thesis [10], where he describes an algorithm *Igor2* that combines analytical and search-based approaches for inductive synthesis. We discuss about it briefly in chapter 5.

Programs

Programmers might sometimes find a programming language as the best means of specifying their intent. Even for applications such as discovery of new algorithms, some people might find it easier to write the specification as an inefficient program rather than a logical relation. We have seen an example of this in section 2.2, where our initial function for finding the minimum of a list of numbers merely picked the head of the list after sorting. Clearly the sorting function used there would take time $O(n^2)$ where n is the size of the input list. This is also the time complexity for the initial algorithm. However, through equational rewriting and unfolding, we derive an efficient program that takes only $O(n)$ time.

Natural Language

Given advances in natural language processing, it is possible to map natural language sentences into logical representations [16]. Natural language can be used as a substitute for logical relations, and end-users might find it very valuable. In particular, natural language interfaces have been designed to query databases to accommodate the need of end-users who interact with databases, but are intimated by the idea of using languages such as SQL. Infact, I myself have worked before on such a project for translating queries from english to SQL given the underlying database schema. There I tried to perform this translation through a deterministic method involving *Context Free Grammars* (CFGs) as well as statistically through a *Machine Translation* system called *Moses*. Here is a link to the project repository `https://github.com/shyamjvs/cs626_project`.

However, the disadvantage with natural language is that it can be ambiguous. More over, from the point of program synthesis, we are not very much interested in this form of user intent as eventually natural language must be converted to a structured (possibly logical) representation for a machine to work with. Thus we choose to neglect natural language inputs, thinking of them as more of the *Machine Translator*'s and *Natural Language Processor*'s job rather than the *Program Synthesizer*'s job.

2.3.2 Search Space

The second dimension in program synthesis is the space of programs over which the desired program will be searched. This choice is made by the developer of the synthesizer and may optionally be restricted by the user of the synthesizer.

The developer of the synthesizer needs to strike a good balance between expressiveness and efficiency of the search space. On one hand, the space of the programs should be large/expressive enough to include programs that users care about. While on the other hand, the space of the programs should be restrictive enough so that it is amenable to efficient search, and it should be over a domain of programs that are amenable to efficient reasoning.

The user of the synthesizer can optionally restrict the search space to obtain programs with specific resource usage. For example, the user might desire a loop-free program, or a program whose memory usage does not exceed a specified amount.

Broadly speaking, the search space can be over the space of (Turing-complete) programs, or restricted form of computational models such as grammars and logics. For the purpose of our work, we choose to neglect computational models involving learning of grammars and logics. The key choice we decide to make is among *Functional* and *Imperative* programs, as these are most widely in use at present.

We choose functional programming over imperative programming due to better suitability to program synthesis and verification. The most significant differences between both stem from the fact that functional programming avoids side effects, which are used in imperative programming to implement state and I/O. Pure functional programming completely avoids side-effects and provides *Referential Transparency*, which makes it easier to verify, optimize, and parallelize programs, and easier to write automated tools to perform those tasks.

Further, higher-order functions are rarely used in traditional imperative programming. Where an imperative program might use a loop to traverse a list, a functional program would use a different technique. It would use a higher-order function that takes as arguments a function and a list. The higher-order function would then apply the given function to each element of the given list and then return a new list with the results. This again is amenable to program verification, as iteration would involve state maintenance. So we achieve recursion (thus iteration) in functional programming, without losing referential transparency.

Chapter 3

Program Derivation in Coq

Programming is the art of designing efficient programs that meet their specifications. There are two approaches to it [15]. The first approach consists of constructing a program and then proving that the program meets its specification. However, the verification of a (big) program is rather difficult and often neglected by many programmers in practice. The second approach is to construct a program and its correctness proof hand in hand, therefore making a posteriori program verification unnecessary.

Program calculation, following the second approach, is a programming technique that derives programs from specifications by means of formula manipulation: calculations that lead to the program are carried out in small steps so that each individual step is easily verified. More concretely, in program calculation, specification could be a program that straightforwardly solves the problem, and it is rewritten into a more and more efficient one without changing the meaning, by application of calculation rules (theorems). If the program before transformation is correct, then the one after transformation is guaranteed to be correct, because the meaning of the program is preserved by the transformation.

In this chapter, we explore the power of Coq as a tool to support program calculation techniques. The authors of [15] were the first ones to identify Coq, which is a popular theorem prover, as a means to implement a powerful system to support program calculation. They have implemented a set of tactics for the Coq proof assistant to write proofs in calculational form. Their work is primarily directed towards developing a theory of lists, and proving it in a calculational and algebraic style.

3.1 Coq: An overview

Coq [1] is a proof assistant which provides an interactive environment for defining objects (integers, sets, trees, functions, programs,...), making statements (using basic predicates and logical connectives), and finally writing proofs. Coq provides user-friendly features required for program derivation techniques, like defining recursive functions, term

[1]`http://coq.inria.fr`

rewriting, polymorphic types, automatic simplification, etc. These features makes it an attractive tool to be used as the underlying system for developing program derivation techniques. An interesting additional feature of Coq is that it can automatically extract executable programs from specifications, as either Objective Caml or Haskell source code.

The Coq proof assistant is based on the *Calculus of Inductive Constructions*(CIC) [4] and implements *Natural Deduction logic system.* This calculus is a higher-order dependent typed λ-calculus [13]. Properties, programs and proofs are all formalized in the same language (CIC). Then, all logical judgments in Coq are typing judgments: the very heart of Coq is in fact a *type-checking* algorithm.

3.2 The language of Coq

Coq objects are sorted into two categories: the *Prop sort* and the *Type sort*:

- *Prop* is the sort for propositions, i.e. well-formed propositions are of type *Prop*. Typical propositions are:

  $\forall\ A\ B : \texttt{Prop},\ A \wedge B \rightarrow A \vee B.$
  $\forall\ x\ y\ :\ Z,\ x \times y = 0 \rightarrow x = 0 \vee y = 0.$

  and new predicates can be defined either inductively, e.g.:

  `Inductive` *even* : $N \rightarrow$ `Prop` :=
  | *even_0* : *even* 0
  | *even_S n* : *odd n* → *even* (*n* + 1)
  `with` *odd* : $N \rightarrow$ `Prop` :=
  | *odd_S n* : *even n* → *odd* (*n* + 1).

  or by abstracting over other existing propositions, e.g.:

  `Definition` *divide* $(x\ y\ :\ N) := \exists\ z,\ x \times z = y.$
  `Definition` *prime* $x := \forall\ y,\ divide\ y\ x \rightarrow y = 1 \vee y = x.$

- *Type* is the sort for datatypes and mathematical structures, i.e. well-formed types or structures are of type *Type*. Here is a basic example of type:

  $Z \rightarrow Z * Z$

  Types can be inductive structures, e.g.:

  `Inductive` *nat* : `Set` :=
  | 0 : *nat*
  | *S* : *nat* → *nat*.

```
Inductive list (A:Type) : Type :=
  | nil : list A
  | cons : A → list A → list A.
```

or types for tuples, e.g.:

```
Structure monoid := {
  dom : Type ;
  op : dom → dom → dom where "x * y" := (op x y);
  id : dom where "1" := id;
  assoc :  x y z, x × (y × z) = (x × y) × z ;
  left_neutral :  x, 1 × x = x ;
  right_neutral :  x, x × 1 = x
}.
```

or a form of subset types called $\sum$-types, e.g. the type of even natural numbers.

$\{n\ :\ N \mid even\, n\}$

Coq implements a functional programming language supporting these types. For instance, the pairing function of type $Z \to Z * Z$ is written $fun\, x \implies (x, x)$ and *cons* (*S* (*S* 0)) (*cons* (*S* 0) *nil*) (shortened to 2::1::nil in Coq) denotes a list of type *list nat* made of the two elements 2 and 1.

Using $\sum$-types, a sorting function over lists of natural numbers can be given the type:

sort : ∀ (*l* : *list nat*), {*l'* : *list nat* | *sorted l'* ∧ *same_elements l l'*}.

where *sorted* is a predicate that expresses that a list is sorted; and *same_elements* says if two lists contain the same elements. On the contrary, a sorting function in a “poor” type system could only be given the following less informative type:

sort : *list nat* → *list nat*

Such a type (specification) enforces the user to write the proofs of predicates *sorted* l' and *same_elements* l l' when writing an implementation for the the function *sort*.

Then, functions over inductive types are expressed using a case analysis:

```
Fixpoint plus (n m:nat) {struct n} : nat :=
  match n with
  | O ⇒ m
  | S p ⇒ S (p + m)
  end
where "p + m" := (plus p m).
```

Coq can now be used as an interactive evaluator. Issuing the command

Eval compute in (43 + 55)

(where 43 and 55 denote the natural numbers with respectively 43 and 55 successors) returns

98 : nat

3.3 Proving in Coq

Proof development in Coq is done through a language of tactics that allows a user-guided proof process. At the end, the curious user can check that tactics build lambda-terms. For example the tactic *intros n*, where *n* is of type *nat*, builds the term (with a hole):

$fun\,(n : nat) \;=>\; \_$

where _represents a term that will be constructed after, using other tactics.

Here is an example of a proof in the Coq system:

```
Inductive seq : nat → Set :=
  | niln : seq 0
  | consn : ∀ n : nat, nat → seq n → seq (S n).
Fixpoint length (n : nat) (s : seq n) {struct s} : nat :=
  match s with
  | niln ⇒ 0
  | consn i _ s' ⇒ S (length i s')
end.
Theorem length_corr : ∀ (n : nat) (s : seq n), length n s = n.
Proof.
  intros n s.
  induction s.
    simpl.
    trivial.
    simpl.
    rewrite IHs.
    trivial.
Qed.
```

Using the Print command, the user can look at the proof-term generated using the tactics:

```
length_corr =
  fun (n : nat) (s : seq n) ⇒
    seq_ind (fun (n0 : nat) (s0 : seq n0) ⇒ length n0 s0 = n0)
      (refl_equal 0)
      (fun (n0 _ : nat) (s0 : seq n0) (IHs : length n0 s0 = n0) ⇒
        eq_ind_r
          (fun n2 : nat ⇒ S n2 = S n0)
          (refl_equal (S n0)) IHs) n s
: ∀ (n : nat) (s : seq n), length n s = n.
```

3.4 Working Principles of Coq Proof System

Theorems are types and their proofs are terms of Coq's calculus. This is based on the well known *Curry-Howard isomorphism* between formal logic and typed lambda calculus. At the level of formulas and types, the correspondence says that implication ($\implies$) behaves the same as a function type, conjunction ($\wedge$) as a "product" or "pair" type, disjunction ($\vee$) as a "sum" or "union" type, the false formula as the empty type and the true formula as the singleton type (whose sole member is the null object). Quantifiers correspond to dependent function space or pairs (as appropriate).

To prove a theorem stated in formal logic, all one needs to do is to show the existance of a λ-term of type corresponding to the theorem. A type is said to be inhabited if there exists a value of that type. A type is only inhabited when it corresponds to a true theorem in mathematical logic.

As an example, consider the *modus ponens* theorem : $P \implies (P \implies Q) \implies Q$. This can be proved by observing that $(\lambda(x : P)(y : P \to Q).yx)$ has type $P \to (P \to Q) \to Q$. Here x and y represent the proof terms of type P and $(P \to Q)$ respectively. (Note that $P \to Q$ is a function type corresponding to the implication $P \implies Q$.) Now if we wish to prove Q, we just need to find and "apply" the proof terms for P and $P \implies Q$ to the above term.

This is how Coq's proof engine works - by constructing a proof term of type corresponding to the given theorem. The Coq system helps the user to build the proof terms and offers a language of tactics to do so. To produce a proof, backward reasoning is used with tactics. A tactice transforms a goal into a set of subgoals (and hence constructing a partial proof tree) such that solving these subgoals is sufficient to solve the original goal. The proof succeeds when no subgoals are left.

Chapter 4

Higher Order Unification

Unification [1], in computer science and logic, is an algorithmic process of solving equations between symbolic expressions. Depending on which expressions (also called terms) are allowed to occur in an equation set (also called *unification problem*), and which expressions are considered equal, several frameworks of unification are distinguished. If *higher-order variables*, that is, variables representing functions, are allowed in an expression, the process is called *higher-order unification*, otherwise *first-order unification*. If a solution is required to make both sides of each equation literally equal, the process is called *syntactical unification*, otherwise *semantical*, or *equational unification*.

4.1 Motivation

Unification, especially higher-order unification, is a technique which is fundamental to the concept of program synthesis. Unification is what links program synthesis to theorem-proving. We had seen earlier in section 2.1, an example of program synthesis (for *max* function of 2 numbers) where the starting point was purely a logical specification. However, in the process of trying to prove the goal (which is essentially a theorem), we found out the required function for calculating maximum. In essence [12], to construct a program meeting a given specification, we prove the existence of an object meeting the specified conditions. However, the proof is restricted to be sufficiently constructive, in the sense that, in establishing the existence of the desired output, the proof is forced to indicate a computational method for finding it. That method becomes the basis for a program that can be extracted from the proof.

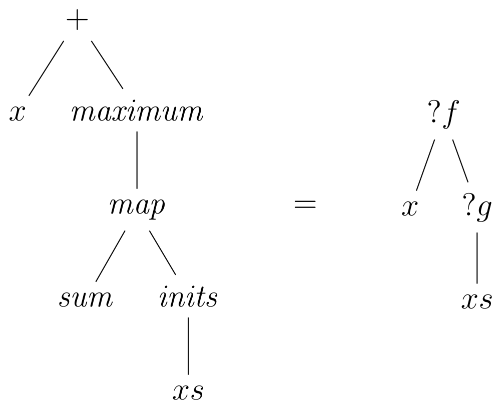
Program synthesis, as we see from a functional programming perspective, is nothing but function computation. We are in essence trying to find a program (function) that satisfies a given (possibly logical) specification. For this we need to find the values of unknown variables in an equation, some of which would be functions. Higher-order unification thus becomes crucial.

[1]parts of this chapter have been adopted from Ankit Gupta's MTP Report

4.2 Illustration

One of the most important steps during a derivation is to instantiate the unknown variables. For example, suppose we are at the following position in the derivation

$$x + \mathit{maximum(map\ sum(inits\ xs))} = ?f\ x\ (?g\ xs)$$



We want to find "values" of $?f$ and $?g$ (pattern variables) , such that the above equation gets "solved". In other words, we seek legitimate "substitutions" for $?f$ and $?g$, which "unifies" the terms on lhs and rhs. The following table shows the possible values for the unknown variables. We assume that we have not fixed any type of $?g$ (and hence of $?f$). We also assume that x and xs are bound variables, and cannot occur freely in substitutions for the pattern variables.

	?f	***?g***
1.	$\lambda a.\lambda b.\ a + b$	$\lambda y.\ \mathit{maximum\ (map\ sum\ (inits\ y))}$
2.	$\lambda a.\lambda b.\ a + \mathit{maximum}\ b$	$\lambda y.\ \mathit{map\ sum\ (inits\ y)}$
3.	$\lambda a.\lambda b.\ a + \mathit{maximum\ (map\ sum\ b)}$	$\lambda y.\ \mathit{(inits\ y)}$
4.	$\lambda a.\lambda b.\ a + \mathit{maximum\ (map\ sum\ (inits\ b))}$	$\lambda y.\ y$

It should be noted that, here we are considering variables which can be instantiated to λ-terms, and not just the ground terms; and equality is with respect to $\alpha\beta\eta$ equivalence. Hence this problem goes beyond the scope of the standard unification algorithm. Owing to this complexity, unlike first order unification, there may not be a canonical choice of the substitution, as shown by the following example.
For $0 = ?f\ ?x$, we can have,

	?f	*?g*
1.	$\lambda a.a$	*0*
2.	$\lambda a.0$	-
3.	$\lambda g.g\ 0$	$\lambda a.a$
4.	$\lambda g.g\ (g\ 0)$	$\lambda a.a$
	...	...

All these matches are incomparable i.e. none of them is a substitution instance of other. It has been shown that the problem of higher order unification is undecidable (Goldfarb [6]). However, Huet and Lang have given an algorithm [9] for a restricted version of this problem, where we are interested only in terms upto *second order*. This is basically a condition on types: a base type (for example int) is first order. The order of a derived type is calculated by adding one to the order of the argument type and taking the maximum of this value and the order of the result type. So for example int $\to$ bool is second order. The order of a term is simply the order of its type. This simple restriction guarantees that there are only a finite number of incomparable matches. In the above example, the matchings which are allowed are 1 and 2.

In the context of functional program derivation and transformation, this restriction is not reasonable, because functions are treated as first class values and can be passed as arguments to other functions. For example, *foldr* :: $(\alpha \to \beta \to \beta) \to \beta \to [\alpha] \to \beta$ is a third order term and routinely used in fusion laws. To overcome this problem, Oege de Moor and Ganesh Sittampalam have given an algorithm for higher order unification [5] in the context of program derivation, which finds more matchings than the standard second order matching algorithm by Huet and Lang. However, they do not claim that the algorithm gives all the third order matchings. Ankit Gupta has studied and implemented this algorithm in Coq. In this chapter, we formally define the problem of higher order unification and a solution/substitution to the hou problem. We further define formally the restricted hou problem specification that Ankit deals with, and the property of the solution obtained.

4.3 The Problem Specification

4.3.1 Definitions

An expression is an untyped λ-term, i.e., a variable , a constant, a λ-abstraction or an application. A variable can be either a *bound* (local) variable or a *pattern* (free) variable (indicated by a preceding ?).

$$E := c \mid x \mid ?p \mid (E_1\ E_2\ ) \mid (\lambda x.E)$$

The notion of "equality" of two expressions is also more complicated as compared to first order matching. Equality is considered modulo

- α - *conversion* : Renaming of bound variables. Hence, the following equality holds

  $$(\lambda x.\ \lambda y.a\ x\ (b\ x\ y)) = (\lambda y.\ \lambda z.a\ y\ (b\ y\ z))$$

- η - *conversion* : Elimination of superfluous arguments. The η-conversion rule states that $(\lambda x.E\ x)$ can be written as E , provided x is not free in E. For example,

  $$(\lambda x.\lambda y.a\ x\ y) = (\lambda x.a\ x) = a$$

  But, the following is not true

  $$(\lambda x.\lambda y.a\ y\ x) = a$$

- β - *conversion* : Substitution of arguments for parameters. An expression of the form $(\lambda x.E_1)\ E_2$ (called as β-redex)is converted to $E_1\ (x := E_2\ )$. The application of this rule in a left-to-right direction is known as β reduction.

An expression is said to be *normal* if it does not contain any η-redex or β-redex as a subexpression. An expression is *closed* if all the variables it contains are bound by an enclosing λ-abstraction. Here, although we are dealing with untyped λ-terms, it can be easily extended to the typed setting, by representing types explicitly in the expressions (as in second order λ-calculus).

A *substitution* is a total function mapping pattern variables to expressions. We shall specify a substitution by listing those assignments to variables that are not the identity. For instance,

$$\varphi = \{\ ?p := (a\ r),\ ?q := (\lambda y.b\ y\ x)\ \}$$

A substitution is said to be *normal* if all expressions in its range are normal, and closed if any variables that it changes are mapped to closed expressions. *Composition* of substitutions φ and ψ is defined by first applying ψ and then φ.

$$(\varphi \circ \psi)\ E = \varphi\ (\psi\ E)$$

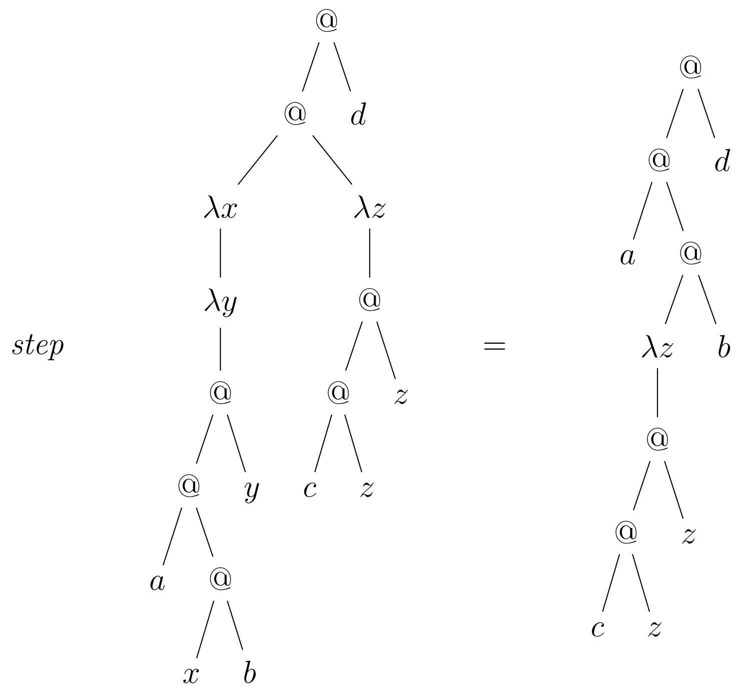
A *rule* is a pair of expressions, written as $(P \to T\ )$, where P does not contain any η-redex, and T is normal, with all variables in T being local variables, i.e. they occur under an enclosing λ-abstraction. The matching process starts off with T closed, but because it proceeds by structural recursion it can generate new rules which do not have T closed. In such a rule, a variable is still regarded as being local if it occurred under an enclosing λ-abstraction in the original rule. We call P the *pattern* and T the *term* of the rule.

4.3.2 Parallel β-reduction

We know define the key operation involved in the specification of the (restricted) higher order unification problem. The function *step* performs a bottom-up traversal of an expression, applying β-reduction wherever possible. Formally,

$$
\begin{aligned}
\mathit{step}\, c &= c \\
\mathit{step}\, x &= x \\
\mathit{step}\, ?p &= ?p \\
\mathit{step}\, (\lambda x.E) &= \lambda x.(\mathit{step}\, E) \\
\mathit{step}\, (E_1\ E_2) &= \mathit{case}\ E_1'\ \mathit{of} \\
&\qquad \lambda x.B \ \rightarrow\ B\,(x \ := \ E_2') \\
&\qquad \ \rightarrow\ (E_1'\ E_2'\,) \\
&\qquad \mathit{where}\ E_1' \ = \ \mathit{step}\, E_1 \\
&\qquad E_2' \ = \ \mathit{step} E_2
\end{aligned}
$$

step proceeds by recursion on the structure of terms and hence always terminates. It is not quite the same as the operation that applies β reduction exhaustively, until no more redexes remain (generally called as β-*normalise*). For example,



In a certain sense, step represents an approximation of β-normalise : if β-*normalise* E exists, then there exists some positive integer n such that step^n E = β-*normalise* E. However, β-*normalise* E does not always terminate.

4.3.3 Restricted HOU problem specification

A substitution ϕ is said to be *pertinent* to a rule $(P \to T\ )$ if all variables it changes are contained in P . Similarly, a substitution is pertinent to a set of rules if all variables it changes are contained in the pattern of one of the rules. A rule $(P \to T\ )$ is *satisfied* by a normal substitution ϕ if

$$\eta - normalise\ (step(\phi\ P))\ =\ T$$

The substitution ϕ is then said to be a *one-step match.* Note that we take equality in the above modulo α-renaming. A normal substitution satisfies a set of rules if it satisfies all elements of that set.

A *match set* M of a set of rules Xs is a set of non-redundant normal substitutions which satisfy Xs (as per the one step match) . The notion of a one-step match contrasts with that of a general match in that it restricts the notion of equality somewhat; a normal substitution ϕ is said to be a general match if

$$\eta - normalise\ (\beta - normalise\ (\phi\ P))\ =\ T$$

It is because of the above restricted specification, that we do not get all possible matches (which may be infinite in number). In the earlier example of $\{?f\ ?x \to 0\}$, the following is a feasible match set :

$$\{\{?f\ :=\ (\lambda\, y.0)\}, \{?f\ :=\ (\lambda\, y.y),\ ?x\ :=\ 0\}\}$$

Clearly,

$$\begin{aligned} step\ ((\lambda\, y.0)\ ?x)\ &=\ 0 \\ step\ ((\lambda\, y.y)\ 0)\ &=\ 0 \end{aligned}$$

Note that, for the substitution $\{?f\ :=\ (\lambda g.g\ 0),\ ?x\ :=\ (\lambda\, y.y)\}$, we have

$$\beta - normalise\ ((\lambda\, g.g\ 0)\ (\lambda\, y.y))\ =\ 0$$

However,

$$step\ ((\lambda\, g.g\ 0)\ (\lambda\, y.y))\ =\ ((\lambda\, y.y)\ 0)\ \neq\ 0$$

Hence this substitution and all other similar substitutions mentioned earlier do not appear in the match set. Now for computing the match set of a given set of rules, we use the algorithm proposed by de Moor and Sittampalam [5]. We do not describe the details of the algorithm in this report, and insist the reader to go through [5] if inquisitive. In essence we wish to make the point that, improvements in unification algorithms is one dimension along which program synthesis can be developed. Since unification is the key technique that introduces constructivism into theorem-proving, the greater the space of functions we search for, the more are the theorems that can be proved.

Chapter 5

Insights from "longest prefix satisfying p" problem

In this Chapter we discuss the problem of finding the length of the longest prefix of a list which satisfies a given arbitrary predicate p. This is a problem that has been studied by Ankit in his project. The derivation of this problem is challenging to automate using the current tactics available in Coq because mechanical simplification alone is not enough to reveal the recursive part. We see why it is so in the derivation ahead. Also, efficient solutions to problems in this class exist only when p satisfies certain conditions. Deriving these conditions required human insights and ingenuity. As a consequence, much of the intelligent parts of the derivation had been carried out by him manually. Our target is to automate, or at least assist the derivation by suggesting useful propositions wherever mechanical simplification gets stuck and is just not enough to take us ahead. We make an interesting connection here with the concept of inductive synthesis, which is essentially performing program synthesis by operating over input-output examples. We observe intuitionally how inductive synthesis can be fused into the deductive program derivation approach, leading to both these techniques working in tandem and complementing each other's weaknesses.

5.1 Problem Specification

For the sake of convenience, we first discuss the derivation of finding the longest prefix itself (and not just its length). We start without assuming any property of p and our end goal is to obtain a linear time solution for the problem. In the process we would derive sufficient conditions on p so as to attain the goal. However, we are not interested in studying the full derivation and focus only on the road blocks that come on our way. For most part of the derivation mechanical simplification suffices. However, as we are going to see ahead, we reach a point where human intervention is required to proceed any further.

To start with, we have the following naive algorithm for solving our problem:

$$\mathit{longest}\ p = \mathit{argmax}\ \mathit{length} \circ \mathit{filter}\ p \circ \mathit{inits}$$

5.2 Derivation

$$\begin{aligned}
longest\,p\,[\,] &= \{\text{definition of } longest\,p\} \\
&\quad argmax\ length\ (filter\,p\,(inits\,[\,])) \\
&= \{\text{definition of } inits\} \\
&\quad argmax\ length\ (filter\,p[[\,]]) \\
&= \{\text{definition of } filter\,p\,[\,] \;=\; true\} \\
&\quad argmax\ length\ [[\,]] \\
&= \{\text{definition of } argmax\} \\
&\quad [\,]
\end{aligned}$$

$$\begin{aligned}
longest\,p\,(x \;:\; xs) &= \{\text{definition of } longest\,p\} \\
&\quad argmax\ length\ (filter\,p\,(inits\,(x \;:\; xs))) \\
&= \{\text{definition of } inits\} \\
&\quad argmax\ length\ (filter\,p\,([\,] \;:\; map\,(x\;:)\,(inits\,xs))) \\
&= \{\text{definition of } filter\,p\,[\,] \;=\; true\} \\
&\quad argmax\ length\ ([\,] \;:\; filter\,p\,(map\,(x\;:)\,(inits\,xs))) \\
&= \{\text{rewrite} : filter\,p \circ map\,f \;=\; map\,f \circ filter\,(p \circ f)\} \\
&\quad argmax\ length\ ([\,] \;:\; map\,(x\;:)\,(filter\,(p \circ (x\;:))\,(inits\,xs)))
\end{aligned} \tag{5.1}$$

As stated before, our efforts during simplification must be directed towards getting the recursive part. However, there is no obvious way to simplify further from this point, since we do not know anything more about p. From here, the idea is to force the dependence on the recursive call and eliminate anything else by imposing restrictions on p.

Let $ys \;=\; longest\,p\,xs$ (the result from the recursive call). Hence,

$$xs \;=\; ys \mathbin{+\!\!+} zs \tag{5.2}$$

for some zs. We use the following identity:

$$inits\,xs \;=\; inits\,(ys \mathbin{+\!\!+} zs) \;=\; (inits\,ys) \mathbin{+\!\!+} (map\,(ys \mathbin{+\!\!+}\,)\,(tail\,(inits\,zs)))$$

which gives us

$$
\begin{aligned}
&= argmax\,length\,([\,]\ :\ map\,(x\ :)\,(filter\,(p\ \circ\ (x\ :))\,(inits\,(ys\ +\!\!+\ zs)))\\
&= argmax\,length\,([\,]\ :\ map\,(x\ :)\,(filter\,(p\ \circ\ (x\ :))\\
&\quad ((inits\,ys)\ +\!\!+\ (map\,(ys +\!\!+\ )\,(tail\,(inits\,zs))))))
\end{aligned}
$$

After this, the derivation proceeds further until we get stuck again at a point where we do not know the value of zs. However, before reaching the next road block, you could have noticed by now that equation 5.2 is an outcome of human insight, and that the derivation stuck at equation 5.1 could move ahead only after feeding in equation 5.2. How could the fact that the answer for a given input list is actually a prefix of it, be made to realize by the synthesizer during the derivation ? This is where inductive synthesis offers hope.

5.3 Help from I/O Examples

Suppose we were given the following input-output pairs as examples for the program,

$$
\begin{aligned}
longest\,p\,[\,] &= [\,]\\
longest\,p\,[1,2,3,4] &= [1,2,3]\\
longest\,p\,[2,3,4] &= [2,3]\\
longest\,p\,[3,4] &= [3]\\
longest\,p\,[4] &= [\,]
\end{aligned}
$$

we could infer from them the fact that the answer ys to $(longest\,p\,xs)$ is actually a prefix of xs. The semantics of the term $prefix$, would assert that xs and ys are related through equation 5.2 for some list zs. In fact, we could go a step further and make assertions which relate the answers of inputs across recursive calls. That is, we could say things like the answer for a given input list $x : xs$ is either the empty list ($[\,]$), or x appended to the answer for the recursive call on xs. That is,

$$
\begin{aligned}
longest\,p\,(x : xs) &= [\,] && if\ (?f\ x\ xs)\\
&= x\ :\ (longest\,p\,xs) && otherwise
\end{aligned}
\tag{5.3}
$$

where $?f$ is a predicate that is to be computed and would involve use of the given predicate p inside its body.

One important question to ask now is how do we come up with a relation like the one in equation 5.2. Clearly, we could keep a database of rewrite rules which relate to properties like list a is a prefix of list b, list a is a suffix of list b, list a is a segment of list b, etc. We could define similar rules for other structures like trees (tree a is a subtree of b), matrices (matrix a is a submatrix of b), etc. Based on the type of the input in our problem, we might look for satisfying properties corresponding to that type one by one from our database, which satisfy the given I/O examples.

5.4 *Igor2* and its Drawback

It would be suitable to talk now about the advantages of this approach over the purely inductive synthesis algorithm *Igor2*. The algorithm for *Igor2* has been proposed by Emmanuel Kitzelmann in [10]. An example synthesis for the *reverse* function for lists has been demonstrated in his thesis. We choose not to mention the algorithm here due to too many technical intricacies, but we encourage the reader to go through the example synthesis for better understanding of it. The algorithm essentially proceeds by iteratively applying 3 kinds of transformations to an initial program that most generally satisfies all the given I/O examples. These tranformations are *rule splitting*, *introducing subfunctions* and *folding back through function calls*. We start by searching through the most general functions for an initial candidate program, and keep partitioning the input examples into groups, each of which satisfy a particular rule, whenever our program fails to simplify further. This is the *rule splitting* transformation. Note that a rule here is a mapping from an input pattern to an output pattern. Secondly, we try to introduce new subfunctions in place of the arguments of the constructor whenever the output pattern of a rule has a constructor at its root. This is the *introduction of subfunctions* transformation. We also generate I/O examples for the new subfunctions we introduced, from the given I/O examples, as we now need to solve for these subfunctions as well. The third kind of transformation, *folding back through function calls*, is what introduces recursion into our program. It essentially folds back the function calls of new subfunctions to function calls of existing functions. This involves mapping the arguments of the new function to the arguments of an existing function in such a way that the outputs remain the same over the given examples. These three transformations are applied iteratively to a working set of candidate programs to obtain the final desired program. Also the programs with a lower degree of rule splitting are explored first in order to favour the synthesis of more general programs over the specific ones.

On deeper inspection of $Igor2$, we notice that even though the I/O examples are used to guide the synthesis of the desired program in the algorithm, it fails to explore certain relations across the examples provided. The final output program is one that satisfies all the I/O examples; but for the sake of the algorithm, all examples are independent upto being grouped within the same rule. This prevents it from relating the output corresponding to a larger input and the output corresponding to a smaller input, that arises as a sub-problem of the larger input. Like we saw in the previous section, the answer for the longest prefix satisfying p for the input list $(x : xs)$ depends on the answer for the smaller list xs through equation 5.3. Inferring such relations is not possible in $Igor2$, as no where does it realize the fact that two inputs are related structurally. Our goal is to enable discovery of such relations in our synthesizer.

Chapter 6

Conclusion

Program synthesis is the task of automatically discovering an executable piece of code given user intent expressed using various forms of constraints such as input-output examples, demonstrations, natural language, etc. We have seen examples of program synthesis starting from different specifications. We have also explored the dimensions along which program synthesis may be carried out. These dimensions are primarily the way user intent is specified, the space of programs we search in and the search technique itself.

Program derivation is a different approach of programming, where we calculate the solution from the problem statement by appealing to correctness-preserving transformations. Functional programming paradigm provides a very neat and elegant platform for expressing problems and developing their solutions by manipulating the expressions using algebraic laws.

The aim is to develop an interactive system which would automate the derivation process, by heuristically applying laws and other techniques and also taking hints from the user for key steps. A system should not only have support for applying these techniques, but also have the intelligence of when/how to apply them. Some of the decisions in the derivation can be tricky and non-obvious, for which the system would rely on users guidance. We wish to reduce such interactions and push the boundaries of capabilities of such a system as far as possible.

We have also studied Coq and found it to be lucrative to be used as the underlying tool for such a system. Its support for automatic rewriting, simplification of expressions, higher order functions, polymorphic types, proof tactics, etc is catalytic for developing program derivation strategies. One major contribution from Ankit has been the implementation of a Coq plugin for higher order unification of terms, based on de Moor and Sittampalam's algorithm, and building a database of useful functional laws for list based functions.

Following this, we studied briefly the derivation for finding the "longest prefix which satisfies p" problem by Ankit. While in his derivation, human insights and ingenuity were used to derive the sufficient conditions on p for a linear time solution, we wish to synthesize even these conditions automatically. For this, we suggest that we make use of input-output examples crafted in such a way that they indicate structural relations between the answer of a larger input and the answer to a smaller input, where the smaller input is a recursive subproblem of the larger input. These insights arise from the case study of an inductive synthesis algorithm *Igor2*. We see that a program derivation technique using algebraic transformation rules lacks ingenuity to introduce new propositions or rules. On the other hand, a purely inductive synthesis technique like that of *Igor2* fails to take advantage of the structural relationships among the examples. In the next phase of the project, we plan to formally concretize this intuition and propose a way of assisting the user in the process of derivation. This would essentially involve suggesting propositions at points in the derivation where we are unable to move ahead.

Bibliography

[1] Umair Z Ahmed, Sumit Gulwani, and Amey Karkare. Automatically generating problems and solutions for natural deduction. In *Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*, pages 1968–1975. AAAI Press, 2013.

[2] Christopher Alvin, Sumit Gulwani, Rupak Majumdar, and Supratik Mukhopadhyay. Synthesis of geometry proof problems. In *Proc. AAAI Conference on Artificial Intelligence*, 2014.

[3] Salman Cheema, Sumit Gulwani, and Joseph LaViola. Quickdraw: improving drawing experience for geometric diagrams. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1037–1064. ACM, 2012.

[4] Thierry Coquand and Gerard Huet. The calculus of constructions. *Information and computation*, 76(2):95–120, 1988.

[5] Oege de Moor and Ganesh Sittampalam. Higher-order matching for program transformation. *Theoretical Computer Science*, 269(1):135–162, 2001.

[6] Warren D Goldfarb. The undecidability of the second-order unification problem. *Theoretical Computer Science*, 13(2):225–230, 1981.

[7] Sumit Gulwani. Dimensions in program synthesis. In *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*, pages 13–24. ACM, 2010.

[8] Sumit Gulwani, Vijay Anand Korthikanti, and Ashish Tiwari. Synthesizing geometry constructions. In *ACM SIGPLAN Notices*, volume 46, pages 50–61. ACM, 2011.

[9] Gérard Huet and Bernard Lang. Proving and applying program transformations expressed with second-order patterns. *Acta informatica*, 11(1):31–55, 1978.

[10] Emanuel Kitzelmann. A combined analytical and search-based approach for the inductive synthesis of functional programs. *KI-Künstliche Intelligenz*, 25(2):179–182, 2011.

[11] Christoph Kreitz. Program synthesis. In *Automated DeductionA Basis for Applications*, pages 105–134. Springer, 1998.

[12] Zohar Manna and Richard Waldinger. Fundamentals of deductive program synthesis. *IEEE Transactions on Software Engineering*, 18:674–704, 1992.

[13] Benjamin C Pierce. *Types and programming languages.* MIT press, 2002.

[14] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. In *ACM SIGPLAN Notices*, volume 48, pages 15–26. ACM, 2013.

[15] Julien Tesson, Hideki Hashimoto, Zhenjiang Hu, Frédéric Loulergue, and Masato Takeichi. Program calculation in coq. In *AMAST*, pages 163–179. Springer, 2010.

[16] Luke S Zettlemoyer and Michael Collins. Learning context-dependent mappings from sentences to logical form. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP: Volume 2-Volume 2*, pages 976–984. Association for Computational Linguistics, 2009.