TECHNISCHE UNIVERSITEIT DELFT

MASTER OF SCIENCE THESIS IN COMPUTER SCIENCE

Program synthesis with dependent types

Luka JANJI Ć

Supervisors:
Dr. Jesper COCKX
Dr. Sebastijan DUMANČIĆ

19th August 2024

Delft University of Technology

Program synthesis with dependent types

Master's Thesis in Computer Science

Programming languages group
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology

Luka Janjić

19th August 2024

Author
Luka Janjić

Title
Program synthesis with dependent types

MSc presentation
21st August 2024

Graduation Committee

Dr. J.G.H. Cockx (chair)	Delft University of Technology
Dr. S. Dumančić	Delft University of Technology
MSc. T. Hinnerichs	Delft University of Technology

Abstract

This thesis investigates the potential of basing a program synthesis system on a dependent type theory. This is an attractive research direction because it allows a very flexible range of specification to be expressed within the same framework. We implement a prototype of a search algorithm driven by unsolved constraints typically generated during dependent type checking. We encode a range of synthesis problems from literature in our system, showcasing how it can be used for expressing the specification and synthesizing programs that manipulate data. We empirically establish the effect of constraint-directed aspect of our approach on performance, based on the encoded problems.

Preface

I would like to express my gratitude to everyone who has supported me throughout the writing of this thesis.

To my parents and my sister, thank you for your encouragement, love, and belief in me.
To my grandma, who was the first to predict my academic path – thank you for your love, prayers, and delicious food.
To my friends, who have stood by me through the ups and downs of this journey, I am grateful for your companionship and the much-needed distractions.
A special thanks to my supervisors, Jesper and Sebastijan. Your guidance, insightful feedback, and advice have been crucial in shaping this work. Thank you for your time and patience.
To Tilman, thank you for always reminding me that time is a real concern and that planning is the only remedy.
And last but not least, to Lola. Thank you for providing love and comfort when it was needed the most. Thank you for keeping it cozy.

To all who have contributed to this journey, whether through direct help with my research, through moral support, or otherwise, I am sincerely grateful. This work would not have been possible without you.

Luka Janjić

Split, Croatia
19th August 2024

Contents

Chapter 1

Introduction

Program synthesis is the process of automatically deriving provably correct programs from a high-level specification, and it has been a long standing holy grail of computer science. The development of efficient synthesis procedures could bring the benefits of programming to a much wider audience, allowing people with little or no programming experience to easily obtain reliable programs.

However, achieving general-purpose synthesis requires searching an infinite space of possible programs and is generally only semi-decidable [Gulwani et al., 2017]. Furthermore, deciding whether a candidate program meets the desired specification is itself a non-trivial problem. The combination of these two factors makes program synthesis a particularly difficult problem to automate: we have no clear way to enumerate possible solutions, nor do we have a clear way to evaluate a single candidate solution.

1.1 Motivation

Typically, the implementation and the specification come from different languages. For example, the implementation language may be a functional programming language such as Haskell, while the specification language may be a first-order logic or simply a set of input-output pairs. A relation between the two must be established to perform the search: we need a way to determine if a candidate program meets the specification. In order for the search to be efficient we want:

- A way to determine what choices are ”obviously wrong” – some choices provably cannot lead to a valid solution.
- A way to propagate necessary conditions for the search to succeed – some choices lead to a valid solution only if certain conditions are satisfied.

A typical synthesis system addresses this by:

- Using an external solver for reasoning in the specification domain

- Dedicated machinery for bridging gap between the specification and implementation; e.g. a translator from the implementation syntax to an encoding in the logic language used by the solver, and a mechanism for enhancing the search by the information provided by the solver.

Because of this, the class of specifications handled by the system is often determined by capabilities of the external solver used, and is often inflexible.

This thesis continues a line of research into type-driven approaches to synthesis – we use the type system of a strongly typed target language as the backbone of our search algorithm. In fact, we chose a language with a particularly expressive type system: a dependently typed language. Using dependent types, we can impose unusually strong requirements on programs that are statically enforced by the type checker. For example, we can write function types that guarantee that their members preserve the size of a data structure that they transform. Then, if we write a function annotated with such type, and the type checker accepts it, we have a guarantee that the required property holds. For example, supposing that Vec n A is defined as the type of vectors of length n and with elements of type A, we may specify the map function as:

$$\text{map} : (A\ B : \text{Type})(n : \mathbb{N}) \to \text{Vec}\ n\ A \to \text{Vec}\ n\ B$$

the function that, for any types A and B, and any natural number n, takes a length n vector of A, and returns a length n vector of B.

This is very attractive from the perspective of program synthesis. If we use dependent types as specification, it solves the problem of verifying candidate programs – it reduces to type checking. Moreover, due to the expressive nature of dependent types, we can express a very broad range of specifications.

1.2 Research questions and contributions

The goal of this thesis is to investigate the prospect of basing a program synthesis system on a dependent type theory. We focus on using unsolved constraints, a typical part of type checking algorithms, in order to propagate information through the search procedure. The goal is to use the associated technique of solving the constraints to prune the search space early. The central idea is that if any constraint becomes violated during the search, we know that the current branch only leads to ill-typed solutions, so we can prune it. We are interested to see to what extent does this integral part of a practical dependent type checker help to guide and prune the search.

Therefore, this thesis is focused on building a synthesis algorithm on top of a dependent type checker, and aims to answer the following research questions:

- **RQ1** Does constraint-guided approach to synthesis of dependently typed programs pay of?

  - **RQ1.1** Does this decrease the time needed to synthesize programs?
  - **RQ1.2** Does this decrease the number of explored states?

- How does this approach compare to other methods of synthesizing functional programs?
  - **RQ2** Can it express more problem specifications than other methods?
  - **RQ3** Can it synthesize programs more efficiently than other methods?

To answer these questions, we have made the following contributions:

1. We implement a small dependently typed language with constraint-based inference and type checking.
2. We implement a dependent type-and-constraint-directed synthesis procedure for this language, to the best of our knowledge, a novel approach.
3. We encode a range of synthesis problems from literature in our system, showcasing how it can be used for expressing specification and synthesizing programs that manipulate data.
4. We empirically establish the effect of constraint-directed aspect of our approach on performance, based on the encoded problems.
5. We compare our procedure with other examples from the literature in terms of synthesis capabilities and expressivity of specification.

The rest of this document is structured as follows:
In chapter 2 we provide the background information and intuitions about program synthesis and dependent types necessary for understanding the rest of the thesis.
In chapter 3 we discuss related work and position our research among the existing approaches to program synthesis.
In chapter 4 we formulate the problem addressed by our research and indicate how we are going to address it.
We present our methods next. First, we introduce the dependently typed language we are using as the basis of our approach in chapter 5. Next, in chapter 6, we introduce the synthesis procedure. Then, in chapter 7, we describe the evaluation methods that we used to answer our research questions.
In chapter 8 we present the results the evaluation, and we discuss them chapter 9.
Finally, we discuss future work and conclude in chapter 10.

Chapter 2

Background

This chapter will explain the prerequisite knowledge required for understanding the contents of the rest of the thesis. We will introduce some terminology and define relevant concepts to aid the exposition in the following chapters. First, we introduce program synthesis, with focus on type-guided synthesis, as it is the basis of this work. Next, we introduce the concept of dependent types and dependently typed languages. We will build up the intuition starting from a simply typed functional language, up to a core language of a dependent type theory. Finally, we will discuss type checking procedures for such languages and the related concept of elaboration.

2.1 Program synthesis

A common approach to synthesis that we adopt is that of iterative refinement of incomplete programs. An **incomplete program** is the one containing **holes**, which stand for an unknown subprogram. We refer to replacing a hole with a more concrete subprogram as **refinement**. The new subprogram may again contain holes but should at least not be a new hole. If a program contains no holes, and thus cannot be further refined, we call it *complete*. Moreover, a complete program is *valid* if it meets the specification. A solution to a synthesis problem then consists of a sequence of refinements that ends in a valid program. For example, the following illustrates a sequence of refinements from an unknown program into a program that doubles the value of x. Question marks represent holes to be completed by refinement, and we mark the one that is refined in red.

$$?_0 \longmapsto ?_1 + ?_2 \longmapsto x + ?_2 \longmapsto x + x$$

2.1.1 Specification

In program synthesis, the specification is the key element that defines what the desired program should achieve. It provides the criteria by which a synthesized program is judged to be correct or incorrect. Depending on the synthesis task,

the specification can take various forms. Three common types are input-output examples, logical specifications, and incomplete programs.

Input-output examples One of the most intuitive forms of specification is a set of input-output examples. Here, the user provides pairs of inputs and their corresponding expected outputs, which serve as a guide for the synthesis process. The goal of the synthesizer is to create a program that correctly maps each input to its respective output. A well known of an example driven synthesis tool is MYTH [Osera and Zdancewic, 2015].

Input-output examples are particularly useful in scenarios where the desired behavior is easy to demonstrate but difficult to describe formally. This approach is common in applications like end-user programming, where non-expert users provide examples to illustrate the behavior they want from a program. However, a significant challenge with this approach is generalization: synthesizing a program that not only works for the provided examples but also generalizes correctly to unseen inputs.

Logical Specification Another way to describe the desired behavior of a program is through logical specification. In this approach, the specification is given as a set of logical formulas that define properties or invariants that the program must satisfy. These formulas can express complex relationships between inputs and outputs that go beyond specific examples, enabling the synthesis of programs whose behavior is difficult to describe precisely with examples. Examples of tool that use a logical specification are LEON [Kneuss et al., 2013] and SYNQUID [Polikarpova et al., 2016]

Logical specifications are particularly well-suited for problems which are more easily formally specified than implemented in a programming language. The main advantage of logical specifications is their precision. However, creating a formal logical specification often requires expertise in formal methods, which may not be accessible to all users.

Incomplete Programs A third form of specification is the use of incomplete programs, also known as sketches or templates. In this approach, the user provides a partial program with holes or placeholders, indicating sections that need to be synthesized. The task of the synthesizer is to fill in these holes. A successful system that works with incomplete programs is SKETCH [Solar-Lezama, 2008]

Incomplete programs strike a balance between the ease of input-output examples and the precision of logical specifications. They allow users to provide structural hints or partial solutions, guiding the synthesis process while leaving details to be automatically filled in. This approach is especially useful when the user has a rough idea of the program structure but is unsure of the exact implementation details.

2.1.2 Issues

The naive way to solve a synthesis problem is to simply enumerate syntactically correct programs until we find a program that meets the specification. While this is very easy to implement, there are serious practical issues with this approach.

Order of enumeration It is possible to enumerate in such a way that we never reach certain programs. For example, if we run a depth-first enumeration, we will never reach most programs. Consider:

$$?_0 \longmapsto ?_1 + ?_2 \longmapsto ?_1 + (?_3 + ?_4) \longmapsto ?_1 + (?_3 + (?_5 + ?_6)) \longmapsto \ldots$$

Validity of specification It will never halt if the specification is contradictory. Even if it is obvious that no program satisfies the specification, we will hopelessly search to no end.

Scaling It does not scale with respect to the complexity of the target language. The number of programs of a fixed size grows exponentially with respect to the number of constructs in our language, and hence does the amount of enumeration we would have to do in the worst case.

The crucial observation is the following: the naive approach would enumerate many obviously faulty programs with respect to the specification. The best we can do is eliminate as many faulty candidates as we can, and enumerate the rest in some "fair" manner.

2.1.3 Type-guided synthesis

We seek to improve on the naive case by pruning the search space. Given an incomplete program, we want to remove any refinements from its domain that are guaranteed not to lead to any valid solutions. One obvious improvement on the purely syntactic approach is to take the type of subprograms into account, yielding **type-guided synthesis**. For example, when synthesizing an argument of addition, it never makes sense to refine it with a logical expression such as $?_i \wedge ?_j$, since this will never yield a valid program. The next step towards reducing the search space is to consider how we can use the specification to rule out more candidate refinements. This is the crux of the problem, and this is what most research is focused on. In type-guided synthesis, we aim to do so by using richer typing, which can impose more restriction on the unknown subprograms.

2.2 Lambda calculus and type systems

Now we turn to dependent type systems. In order to understand dependent types, we first need to understand some more fundamental concepts, namely pure func-

tional programming and type systems. In section we aim to build up the intuition behind these building blocks, culminating in the exposition of dependent types.

2.2.1 Pure functional programming and lambda calculus

Functional programming is a paradigm in which functions are the core building blocks. A language is considered pure if it only allows *pure functions*, meaning functions that produce output solely based on their input. This requires immutability of variables and persistent data structures. For instance, adding an element to a list creates a new list with the added element, leaving the original unchanged. As a result, evaluating pure functional programs involves substituting function arguments until no further substitutions are possible. The languages discussed in this thesis follow this approach.

The simplest pure functional programming language is the **untyped lambda calculus** $\mathcal{L}_\lambda$, the language consisting of only function abstractions, function applications and variables:

Variables	V	$::= x, y, z, ...$	
Terms	L	$::= \lambda V.L$	abstraction
		$\mid L\ L$	application
		$\mid V$	variable

In order to avoid issues later on, we will assume that every abstraction introduces a unique variable. Furthermore, we assume a notational convention in which the function abstraction associates and binds maximally to the right, whereas the application associates to the left. Thus, for example, we write $\lambda x.\lambda y.\lambda z.xyz$ and mean $\lambda x.(\lambda y.(\lambda z.((xy)z)))$. Moreover, if we have an uninterrupted sequence of abstractions we will often omit the λ symbols of all but the first abstraction and only write the variable names in sequence. With this, the above example becomes simply $\lambda xyz.xyz$.

Computation in $\mathcal{L}_\lambda$ amounts to repeated application of a rule called β**-reduction**. This rule simply states that an application of a function to some expression "reduces" to the result of substituting that expression into the function's body. We denote β-reduction as a relation $\rightarrow_\beta$ and we read $S \rightarrow_\beta T$ as "S β-reduces to T in one step". The notation $S[T/x]$ stands for the result of substituting every free occurrence of x in S by T. We call any expression of the form $(\lambda x.S)T$ a (β-)*redex*. A term that contains no redexes is maximally reduced and can be seen as the result of a computation. Such a term we call a **normal form**.

Now that we have a notion of computation, we may ask ourselves: "When are two $\mathcal{L}_\lambda$ expressions equal?" Consider the following two expressions:

$$\lambda fx.fx \quad \lambda hy.hy$$

Clearly, which ever two expressions N, M we apply to them, the result will be the

same:

$$(\lambda fx.fx)NM \to_\beta (\lambda x.Nx)M \to_\beta NM$$
$$(\lambda hy.hy)NM \to_\beta (\lambda y.Ny)M \to_\beta NM$$

The general observation we illustrate with this example is that two expressions that differ only in terms of names of their bound variables are computationally indistinguishable. We call the act of renaming all occurrences of a bound variable by a fresh variable α**-conversion**, and call any two terms that are the same up to α-conversion α-convertible, writing this as $=_\alpha$. Now consider a different pair of expressions:

$$(\lambda xy.xy)(\lambda z.z)(\lambda w.w)$$
$$(\lambda zw.w)(\lambda x.x)$$

At first glance, these two expressions are rather different. Nevertheless, let us see what happens when we β-reduce them:

$$(\lambda xy.xy)(\lambda z.z)(\lambda w.w) \to_\beta (\lambda y.(\lambda z.z)y)(\lambda w.w) \to_\beta (\lambda z.z)(\lambda w.w) \to_\beta \lambda w.w$$
$$(\lambda zw.w)(\lambda x.x) \to_\beta \lambda w.w$$

They reduce to the same normal form! Whenever two expressions S, T reduce to the expression, we call them β-convertible and write $S =_\beta T$. A famous result from the early days of computer science tells us that for any two ways we reduce the same term, the results will always be β-convertible [Church and Rosser, 1936]. We call this confluence. A consequence of this fact is that a term can have *at most one* normal form. Indeed, some terms have no normal form:

$$(\lambda x.xx)(\lambda x.xx) \to_\beta (\lambda x.xx)(\lambda x.xx)$$

Finally, consider the following pair of expressions: $\lambda x.(\lambda y.y)x$ and $\lambda y.y$. As with α-convertibility, which ever expression M we apply to these terms, they compute to the same result:

$$(\lambda x.(\lambda y.y)x)M \to_\beta (\lambda y.y)M \to_\beta M$$
$$(\lambda y.y)M \to_\beta M$$

Generally, we identify any term T with the term $\lambda x.Tx$, since they are computationally indistinguishable. We call this notion η**-convertibility**, and write $T =_\eta \lambda x.Tx$.

2.2.2 Evaluation strategies

When implementing an evaluator for a functional language, it is crucial to define an appropriate **evaluation strategy**. In lambda calculus, this involves determining the order in which subterms of an application are evaluated.

A common approach is the *lazy evaluation strategy*, or call-by-name. This strategy delays the evaluation until necessary, which can prevent unnecessary computations and avoid divergent reductions [Barendregt et al., 1984]. However, lazy evaluation can result in redundant evaluations of the same expression. To address this, practical implementations often use *call-by-need*, which memoizes the result of the first evaluation of an expression, avoiding duplicated computations.

Another key concept is **weak head normal form** (WHNF). Rather than fully reducing a term, evaluation stops once an unapplied function is reached, without delving into the function body. Although WHNF is not suitable for β-equality comparisons, it minimizes computation and is useful in practic.

To formalize evaluation strategies, we often define an evaluation relation. For example, **big-step evaluation** focuses on directly obtaining the result of a computation. We denote lazy evaluation to WHNF as $\Downarrow$, where $s \Downarrow t$ means that the term s evaluates to the WHNF t.

2.2.3 Adding a simple type system

Now, we motivate and introduce the notion of a **type system**, through the example of a typed version of $\mathcal{L}_\lambda$, the **simply typed lambda calculus** $\mathcal{L}_\to$. For computation to be meaningful and ultimately useful, we need a way to encode problems. To encode problems we have to represent the *givens*, the data, and the corresponding notions of manipulating it. For now, suppose we extend the lanugage with some primitive notions of data, such as numbers; some arithmetic operation, say addition, to manipulate them; and we extend our β-reduction to evaluate arithmetic expressions in the usual way.

$$\begin{array}{rll} L ::= & ... & \text{previous definition} \\ | & n \in \mathbb{N} & \text{numbers} \\ | & L + L & \text{addition} \end{array}$$

With this, we can perform some computation:

$$29 + 13 \to_\beta 42$$
$$(\lambda x.x + x)2 \to_\beta 2 + 2 \to_\beta 4$$

However, a problem arises in that not all data can be manipulated in the same way. We can add and multiply numbers, but not functions. We cannot apply numbers to terms. We can meaningfully apply *some* functions to numbers, but not others:

$$(\lambda x.x + x)2 \to_\beta \ldots \to_\beta 4$$
$$(\lambda f.(f1))2 \to_\beta 2(1) \to_\beta ??$$

Syntax is not restrictive enough to rule out meaningless terms. Thus, we want to categorize terms based on the kind of data they compute. A way standard way to

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A}\ \textsc{Var} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda\, x\, .\, t : A \to B}\ \textsc{Lam} \qquad \frac{\Gamma \vdash t : A \to B \quad \Gamma \vdash s : A}{\Gamma \vdash t\, s : B}\ \textsc{App}$$

$$\frac{}{\Gamma \vdash n : \mathbb{N}}\ \textsc{Nat} \qquad \frac{\Gamma \vdash s : \mathbb{N} \quad \Gamma \vdash t : \mathbb{N}}{\Gamma \vdash s + t : \mathbb{N}}\ \textsc{Plus}$$

Figure 2.1: A simple type system for $\mathcal{L}_{\to}$

achieve this is by means of a **type system**. First, we define the syntax for expressing types:

$$\begin{array}{lrll} \text{Types} & T & ::= \tau \in \mathcal{B} & \text{base type} \\ & & \mid T \to T & \text{function type} \end{array}$$

where $\mathcal{B}$ is the set of base types; types of the primitive data. In the current example, this set contains only the type $\mathbb{N}$. We assign types to terms, expressing this by means of type annotations $t : \tau$ for some term t and type τ. Extending $\mathcal{L}_{\lambda}$ with annotations and types, we obtain the **simply typed lambda calculus** $\mathcal{L}_{\to}$.

Of course, we only want to validate certain type assignments such that they correspond to the meaningful categorization of terms. An assertion that a term t has a type τ is called a **typing judgement**, and we write it as $\vdash t : \tau$. Often we want to assert typing under a set of assumptions $\{x_i : \sigma_i\}_i$, stating that each variable x_i is of type σ_i. A collection of such assumptions we call a **context**, often using the symbol Γ to represent an arbitrary set of assumptions. For consistency, we will assume that contexts assign at most one type to a variable, so we can also think of them as partial functions from variables to types, and write $\Gamma(x)$ to refer to the type that Γ assigns to x, if such a type exists. Then, for a typing judgement with a context of assumptions Γ we write $\Gamma \vdash t : \tau$, and interpret $\vdash t : \tau$ as the special case $\emptyset \vdash t : \tau$.

A **type system** defines the categorization of terms into types by defining the typing relation $\vdash$. Typically this is done in a syntax directed manner, such that we assign a type to each primitive language construct, and provide rules for assigning types to composite expressions conditioned on the types of their immediate subexpressions. With a type system in place, we can filter out many meaningless expressions *statically*; without needing to execute them. If a type can be assigned to a term then the term is guaranteed to have some meaning. The strength of this guarantee coincides with the strength of the type system, while the range and specificity of the meaning we can describe corresponds to the expressivity of the system.

$$\dfrac{\dfrac{\dfrac{(x:\mathbb{N}) \in \{x:\mathbb{N}\}}{\{x:\mathbb{N}\} \vdash x:\mathbb{N}}\,\textsc{Var} \quad \dfrac{}{\{x:\mathbb{N}\} \vdash 1:\mathbb{N}}\,\textsc{Nat}}{\{x:\mathbb{N}\} \vdash x+1:\mathbb{N}}\,\textsc{Plus}}{\vdash \lambda\, x\, .\, x+1 : \mathbb{N} \to \mathbb{N}}\,\textsc{Lam}$$

Figure 2.2: An example derivation of a typing judgement

2.3 Dependent types

In this section, we transition from a simply-typed to a dependently-typed language. A classic example involves lists of fixed sizes. Instead of categorizing all lists under a single type, we refine the classification by length, assigning different types to lists of various lengths. Although this could be done by defining separate types like $List_0$, $List_1$, and so on, this approach quickly becomes impractical.

The dependently typed approach offers a solution. It generalizes the function type to the **dependent function type** and removes the distinction between types and values, allowing more expressive and flexible type definitions. We reflect this in the grammar of our language $\mathcal{L}_\Pi$:

Variables	V	$::= x, y, z, ..., \alpha, \beta, \gamma, ...$	
Terms, types	T	$::= \lambda V.T$	abstraction
		$\mid T\ T$	application
		$\mid (V:T) \to T$	dependent function
		$\mid$ Type	universe
		$\mid V$	variable

In dependently typed languages, the lack of distinction between types and values allows for much more expressivity in the type system. For example, we can define a type that depends on a specific value, such as a list type that encodes its length directly in the type itself. Let us call this type Vect. A vector with three elements of type A would have the type Vect$3A$, where 3 is a value that appears at the type level. This allows us to enforce constraints, such as ensuring that mapping a list preserves its length. This is expressed in the type of the map function on Vect:

$$\text{map} : (A\ B : \text{Type})(n : \mathbb{N}) \to \text{Vec}\ n\ A \to \text{Vec}\ n\ B$$

To build vectors, we have two constructors, namely nil and cons, and they have the following types:

$$\text{nil} : (n : \mathbb{N})(A : \text{Type}) \to \text{Vec}\ n\ A$$
$$\text{cons} : (n : \mathbb{N})(A : \text{Type}) \to A \to \text{Vec}\ n\ A \to \text{Vec}\ (\text{succ}\ n)\ A$$

where succ is the successor function on the natural numbers.

2.3.1 Recursors

A mechanism for defining functions on data types is called a **recursor**. A recursor is itself a function that embodies the recursion principle on the type. The idea behind recursors is that we provide one function per constructor, and then the recursor applies them uniformly throughout the structure, reducing it to a value. The type into which we reduce it is called the **motive**. Let us look at examples. If we consider natural numbers as a data type with two constructors, namely 0 and succ , then the recursor for natural numbers looks like this:

$$\text{natrec} : (A : \text{Type}) \to A \to (A \to A) \to \mathbb{N} \to A$$

With it, we can define addition as follows:

$$\text{add} = \lambda\ n\ m\ .\ \text{natrec}\ \mathbb{N}\ n\ \text{succ}\ \ m$$

This tells us that we should take m, replace succ with itself and 0 with n. Thus, if we had $n = \text{succ (succ zero)}$ and $m = (\text{suc zero})$, we would obtain the result succ (succ (succ zero))

The following is the type of the recursor on vectors:

$$\begin{aligned}\text{foldVec} : (A : \text{Type})(n : \mathbb{N})(P : \mathbb{N} \to \text{Type})\\ \to P\ 0 \to ((m : \mathbb{N}) \to A \to P\ m \to P\ (\text{succ}\ m))\\ \to \text{Vec}\ A\ n \to P\ n\end{aligned}$$

As type of vectors is indexed, so is the motive in its recursor. With the recursor, we can sum up the entries in a vector of n naturals:

$$\text{foldVec}\ \mathbb{N}\ n\ (\lambda\ x\ .\ \mathbb{N})\ 0\ (\lambda\ m\ x\ s.\ x + s) : \text{Vec}\ \mathbb{N}\ n \to \mathbb{N}$$

The function for the cons case takes the current value x and the sum of the rest of the values s and adds them together. Generally, the motive for an indexed data type will have the same indices as the type.

A more powerful notion than recursors is the induction principle. While it has the same computational meaning, it has a more general motive. It allows us to write inductive proofs on our data, as seen through the Curry-Howard correspondence [Curry, 1934][Howard et al., 1980].

2.4 Dependent type checking

Dependent type checking is the process of verifying that expressions conform to the expected types in a dependently typed language. The expressiveness of these systems comes with additional complexity in the type-checking process.

To manage this complexity, many dependent type systems use a technique called **bidirectional type checking**. In bidirectional type checking, the system alternates

between two modes: type inference and type checking. In simpler systems, such as Hindley–Milner [Hindley, 1969][Milner, 1978], the type checker can infer the type of any expression. However, in dependently typed languages, the additional expressiveness means that bidirectional type checking is crucial to handle the complex interactions between types and values.

2.4.1 Unification

Recall that arguments such as the length and type of the content of a vector had to be passed explicitly along with it to any function. This quickly becomes cumbersome and clutters the code. To address this issue, dependent type systems often support *implicit arguments*, which are automatically inferred by the type checker.

Implicit arguments are solved by a process called **unification**. Unification is a process for solving equations between terms with holes in them. In this context, the holes are called meta variables. Before type checking, during a process called elaboration, a type checker will place meta variables for all implicit arguments. Then, during bidirectional type checking, equations are generated whenever the system both expects a type for a term, and can infer it. In the end, the solution is a **substitution** mapping meta variables to terms, that satisfies all the equations.

Depending on the power, a unification algorithm can be higher-order or first-order, depending on whether functions can or can not be solutions to meta variables. In first-order unification the process of solving equations is well-understood and efficient, while solving higher-order equations is undecidable in general. Modern dependently typed languages implement sophisticated algorithms for solving decidable fragments of higher-order unification. These mechanisms help to make dependently typed languages practical for real-world use by automating tedious tasks that would otherwise be left to the programmer.

2.5 Encoding datatypes

We have been discussing various data types despite the fact that we never introduced a way to define them. This is because, in a dependently typed language, we can *encode* it using only the constructs in $\mathcal{L}_{\Pi}$. The idea behind this lies in representing data types by their recursion principles. For example, we encode the natural numbers as the type

$$(A : \text{Type}) \to A \to (A \to A) \to A$$

and vectors as the type

$$\begin{aligned}(A : \text{Type})&(n : \mathbb{N})(P : \mathbb{N} \to \text{Type}) \\ &\to P\ 0 \to ((m : \mathbb{N}) \to A \to P\ m \to P\ (\text{succ}\ m)) \\ &\to P\ n\end{aligned}$$

Ideally, we would represent data types by their induction principle since it is more general, however, it is known that this is not possible in a standard dependent type theory [Geuvers, 2001].

The main motivation for encoding data instead of introducing a language construct for defining them is the ease of implementation. However, this comes at the cost of usability and power.

2.6 Encoding specification

As we mentioned earlier, types can be interpreted as logical propositions. Correspondingly, programs belonging to those types can be seen as proofs of those propositions. For example, we can interpret a dependent function type as universal quantification, and a non-dependent function type as implication.

One useful dependent type that we have not yet introduced is the **dependent pair** or the **sum** type $\Sigma\ A\ B$. The sum type generalizes the usual pair type such that the type of the second element depends on the value of the first. We can see this in its type: $(A : \text{Type})(B : A \to \text{Type}) \to \text{Type}$. By Curry-Howard correspondence relates the sum type to the *existential quantifier*. It is encoded by:

$$\Sigma = \lambda\ A\ B\ .\ (X : \text{Type}) \to ((x : A) \to (B\ x) \to X) \to X$$

Another ubiquitous dependent type is the **identity type**, which corresponds to the equality relation. The identity type is indexed by two values of any type A and is populated if and only if the two indices are the same. We encode it with the Leibniz identity [Leibniz, 1686]:

$$\text{Id} = \lambda\ A\ x\ y.\ (P : A \to \text{Type}) \to (P\ x) \to (P\ y)$$

For convenience, we will write $s \equiv t$ for Id $A\ s\ t$ where it is clear that s and t are of type A.

With this in mind, we can represent logical propositions such as $\forall x\ y \in \mathbb{N}\ .\ x = y \implies y = x$ by types such as $(x\ y : \mathbb{N}) \to x \equiv y \to y \equiv x$. Moreover, we can represent IO examples that a program must specify using Σ and Id. For example we can specify a function that maps 1 to 2 and 2 to 4:

$$\Sigma\ (\mathbb{N} \to \mathbb{N})\ (\lambda\ f\ .\ f\ 1 \equiv 2 \wedge f\ 2 \equiv 4)$$

Chapter 3

Related work

Research in program synthesis is abundant, and various flavours of type driven synthesis have been successfully explored. In this chapter, we discuss some past work in type driven program synthesis. We first explore some well-known solutions that exploit specific forms of specification. Next, we discuss the work done in synthesis in dependently typed context. Finally, we identify the gap in the research on synthesis with dependent types.

3.1 Type driven program synthesis

An early effort in type driven synthesis is Djinn [Augustsson, 2014], a synthesis system for Haskell, implemented in Haskell. It takes polymorphic types of Haskell programs and uses the Curry-Howard correspondence to transform the problem of synthesizing a Haskell program into a problem of constructing a proof in a decidable logic. It is straightforward and works well for simple programs, however, polymorphic types are too general and often fit a too wide range of program.

In a simply typed context, the type of the target program is typically not descriptive enough to serve as the specification on its own and various methods have been developed to integrate additional specification into the search. One way to further refine the specification is by giving IO examples, as done by Osera and Zdancewic [Osera and Zdancewic, 2015]. The authors develop a prototype implementation called MYTH, which synthesizes programs in a simply typed functional language with algebraic data types. The crucial observation made in this work is that the constraints derived from examples can be propagated down the incomplete program to the holes and used to partially evaluate it before any complete program is reached. This enables ruling out incomplete programs that could never satisfy the specification before any concrete instance is enumerated.

While examples are an intuitive and simple way of stating a specification simple monomorphic programs, their expressivity is limited. A more expressive form specification is needed to concisely express more complex properties. LEON is

another system targeting simply typed functional programs [Kneuss et al., 2013]. In LEON, the specification is expressed in first-order logic, and the system uses an SMT solver to verify candidate programs. If a candidate is rejected, the solver returns a counterexample, which is then used to guide the search further. However, LEON does not decompose the specification to narrow down the search for subprograms.

A particularly interesting example for our work is SYNQUID, presented in [Polikarpova et al., 2016]. The form of specification used in this work is *polymorphic refinement types* – polymorphic types decorated with a predicate from some fixed logic. The predicate is called a refinement[1], and the result of decorating a base type with a refinement predicate is a refinement type. They can be seen as a special case of the dependent sum type. For example, using the notation from the paper we can represent the type of even integers as $\{\text{Int} \mid \nu \mathbin{\%} 2 = 0\}$. The equivalent dependent type would be $\Sigma\ \text{Int}\ (\lambda x.x \mathbin{\%} 2 \equiv 0)$. The main difference is that refinement predicates are restricted to be decidable by an SMT solver [Rondon et al., 2008]. The major technical contribution of this work is a synthesis system based on refinement type checking and inference. Exploiting a specialized solver, refinement predicates of yet-to-be-synthesized subprograms can be determined. The inference step is crucial because it enables decomposing the specification of the current hole over the subgoals, propagating information down the search tree, and ultimately cutting down the search space.

3.2 Synthesis in dependently typed languages

We turn to past work on synthesis in a dependently typed context. Some past works in this area focused mainly on theoretical results, but were not intended to be implemented for a programming language. For example, Dowek has demonstrated a complete synthesis procedure for a dependent type theory [Dowek, 1993]. This provided foundations for later implementations aimed at simplifying programming in dependently typed languages.

An early example of such work is in the context of the programming language Agda [Norell, 2007]. In the early days of the language, the tool Agsy [Lindblad and Benke, 2004] has been integrated with its standard interactive development environment. It was designed with automated theorem proving in mind, rather than general program synthesis, specifically aimed at simple intermediary lemmas. The algorithm implements a type-guided search among the possible derivations, employing iterative deepening with rule-based reasoning. The system heuristically enumerates the program space using the available components, searching for a well-typed term in the order of size of the derivation. Crucially, constraints are not part of the search state. Thus, a refinement is accepted only if unification succeeds

[1] Due to the unfortunate clash of terminology, and to avoid confusion, I shall refer to the predicates in refinement types as refinement predicates, while referring to refinements of incomplete programs in program synthesis simply as refinements.

without unsolved constraints.

Recently, a new synthesis tool called Mimer [Skystedt, 2022] has been developed for Agda, intended to replace Agsy. Its primary design goal was tighter integration with Agda's implementation compared to that of Agsy. As such, it addressed the issue of integration but did not improve the synthesis capabilities of Agsy. While the author demonstrated broader synthesis capabilities of Mimer, covering features predated by Agsy, it is also significantly slower and lacking support for some fundamental features such as case splitting, preventing it from synthesizing many data-manipulating programs. In contrast with Agsy, Mimer keeps constraints as part of the search state, collecting any unsolved constraints produced during refinement. Once it reaches a complete candidate program, it attempts to unify the collected constraints and accepts the candidate if it succeeds.

An orthogonal line of work to Agsy and Mimer strives to use dedicated theory solvers for proof automation. Kokke and Swierstra develop a Prolog style procedure for in Agda *in Agda* using metaprogramming [Kokke and Swierstra, 2015]. Kokke implemented an Agda library called Schmitty [Kokke] for embedding SMT querries into Agda. Similarly, Foster and Struth integrated an equational solver into Agda [Foster and Struth, 2011].

3.3 The gap

The overarching theme of MYTH and SYNQUID is propagating the consequences of refinements down the search tree in order to prune the search. The success of their approaches indicates that there is value in investing additional effort to reason about the the current incomplete program with respect to the specification, and incorporate the obtained knowledge into the search.

The existing body of work on synthesis with dependent types that we have discussed fits into one of two categories. Either the focus is on integration with solvers, or it is on different techniques of enumerating the program space: the question of reducing the search space is left open. Both Agsy and Mimer do no intermediary work while exploring the search space. Moreover, in this regard, they can be seen as existing on the opposite sides of a spectrum. Agsy naively discards any inconclusive refinements. Thus, it prunes too much, discarding potentially fruitful refinements. On the other hand, Mimer accepts inconclusive refinements, but it does not look back until a concrete candidate is found. As such, it explores too much since some constraints may become violated after they are emitted but before a concrete candidate is reached.

The gap between these two sides of the spectrum remains unexplored, and this inspired the research that is presented in the rest of this thesis.

Chapter 4

Problem description

This chapter describes the problem addressed by the research in this thesis and presents the core idea of the proposed solution.

4.1 Naive approach

In a broad sense, the goal of this thesis is to explore the suitability of a dependently typed language as a basis for a program synthesis system. However, there is a fundamental challenge to building a synthesis system on top of a dependently typed language. Incomplete programs may be conditionally type correct, given that constrains can later be satisfied. Naively, this could be handled in one of two ways:

1. When checking a candidate refinement, unify component's type with that of a meta. If it succeeds, accept the refinement. If it fails or produces unsolved constraints, discard the refinement.

2. If there are unsolved constraints emitted, collect them. Once a complete candidate program has been constructed, run unification on all the collected constraints.

In the first case, many infeasible programs are discarded early, however, so are many feasible ones. In the second case, we have the opposite problem: many infeasible programs end up enumerated, despite the fact that their infeasibility may have become apparent much earlier.

4.2 Constraint guided approach

The goal of this thesis is to offer a middle ground solution to the said problem. We want to track any unsolved constraints associated with a search branch and prune branches from the search tree as soon as it becomes clear that they cannot lead to a well-typed solution. Furthermore, if they become solvable, they might imply solutions to some unsolved meta variables. Let us illustrate the benefits

of constraint guided approach by two examples. Consider the following example based specification:

$$\Sigma\ (\text{Bool} \to \text{Bool} \to \text{Bool})\ \lambda\ f\ .\ (f\ \text{true false} \equiv \text{false}) \wedge (f\ \text{false true} \equiv \text{true})$$

The first argument to "if-then-else" is the return type – the motive – which is also the expected type of the remaining two arguments: the "then" and the "else" branches.

First, consider what happens when refining the following incomplete program:

$$(\lambda\ p\ q\ .\ \text{ite Bool}\ ?_0\ ?_1\ p\ ,\ \text{refl}\ ,\ \text{refl})$$

The constraints that we have gathered now look like this:

$$\text{ite Bool}\ ?_0\ ?_1\ \text{true} \stackrel{?}{=} \text{false}$$
$$\text{ite Bool}\ ?_0\ ?_1\ \text{false} \stackrel{?}{=} \text{true}$$

Suppose that the next meta we refine is $?_0$, and we consider refining it with the value false. We successfully unify the type of false with the type of $?_0$ – both are Bool. Before we accept the refinement and continue the search, we *propagate* – run unification on the tracked constraints in order to check if any became solvable or violated – and we find that the first has been violated:

$$\text{ite Bool false}\ ?_1\ \text{true} \stackrel{?}{=} \text{false} \implies \text{true} = \text{false}$$

Therefore, we stop the search there, because we know that no substitution of $?_1$ can make the resulting program well-typed.

Now, consider what happens in the search branch that has reached the following incomplete program:

$$(\lambda\ p\ q\ .\ \text{ite Bool}\ ?_0\ ?_1\ ?_2\ ,\ \text{refl}\ ,\ \text{refl})$$

In this state, the constraints induced by the examples are

$$\text{ite Bool}\ ?_0\ ?_1\ ?_2 \stackrel{?}{=} \text{false}$$
$$\text{ite Bool}\ ?_0\ ?_1\ ?_2 \stackrel{?}{=} \text{true}$$

Suppose, again, $?_0$ is selected to be refined next, but the candidate for refinement is the variable p. We successfully unify the types and propagate:

$$\text{ite Bool true}\ ?_1\ ?_2 \stackrel{?}{=} \text{false} \implies ?_1 = \text{false}$$
$$\text{ite Bool false}\ ?_1\ ?_2 \stackrel{?}{=} \text{true} \implies ?_2 = \text{true}$$

Now, both constraints have now become solvable, and unifying them implies the solution to both remaining meta variables.

Chapter 5

Language

This section presents the dependently typed language with meta variables $\mathcal{L}^{?}_{\Pi}$, the target and the specification language of our synthesis procedure. First we introduce the syntax of the language. Then, we define the evaluation, describe the unification and finally, we present the type system.

5.1 Syntax

The language is a small dependent type theory. The grammar is extended from that of $\mathcal{L}_{\lambda}$ to include let abstraction, dependent function type, universe type and meta variables. The syntax of the language is shown in 5.1. The the constructs and their syntax are standard for a dependently typed language.

The surface syntax also includes *holes*, represented by underscores. They are intended to be inserted by the user, as an indication of what is expected to be filled in by synthesis. They are removed during type checking, which outputs a modified syntax tree with each hole replaced by a fresh meta variable.

5.2 Evaluation

Checking dependent types requires checking the equality of terms they depend on, which evaluation is an indispensable part of. The straightforward approach is to compute normal forms of two terms and check if the two are α and η equivalent. For sake of performance, we take a lazy approach: our evaluation procedure reduces terms to their WHNF, doing only as much computation as is necessary to start working with the expression, deferring unnecessary computation [Howe, 1989].

We present the big-step evaluation rules in 5.2. In the EAppNtr rule, we use the predicate on terms Neutral$(\cdot)$. This predicate is true if the given term is a variable, application or a meta variable. We use it to disambiguate the lambda case.

Variables	V	$::=$ $x, y, z, ..., \alpha, \beta, \gamma...$	
Terms	T	$::=$ $\lambda V.T$	abstraction
		$\mid$ $T\ T$	application
		$\mid$ V	variable
		$\mid$ $?_i$	meta variable
		$\mid$ $V := T; T$	let abstraction
		$\mid$ $(V : T) \to T$	dependent function type
		$\mid$ $T : T$	annotated term
		$\mid$ Type	universe

Figure 5.1: The grammar of $\mathcal{L}^{?}_{\Pi}$

$$\frac{}{\text{Type} \Downarrow \text{Type}}\ \textsc{EUniv} \qquad \frac{}{\lambda x.t \Downarrow \lambda x.t}\ \textsc{ELam} \qquad \frac{}{x \Downarrow x}\ \textsc{EVar} \qquad \frac{}{?_i \Downarrow ?_i}\ \textsc{EMeta}$$

$$\frac{s \Downarrow \lambda x.s'}{s\ t \Downarrow s'[t/x]}\ \textsc{EAppLam} \qquad \frac{s \Downarrow s' \quad \text{Neutral}(s')}{s\ t \Downarrow s'\ t}\ \textsc{EAppNtr}$$

$$\frac{\tau \Downarrow \sigma}{(x : \tau) \to \tau' \Downarrow (x : \sigma) \to \tau'}\ \textsc{EPi}$$

$$\frac{t \Downarrow v}{t : \tau \Downarrow v}\ \textsc{EAnn} \qquad \frac{t[s/x] \Downarrow v}{x := s; t \Downarrow v}\ \textsc{ELet}$$

Figure 5.2: Big-step evaluation for $\mathcal{L}^{?}_{\Pi}$ computing WHNF

5.3 Unification

Both type checking and synthesis make heavy use of unification. The purpose is twofold. The first and foremost is comparing terms for equality, crucial for dependent type checking. The second is automatically inferring uniquely implied solutions to meta variables, corresponding to inference of implicit arguments in dependently typed languages [Pollack, 1990]. While the latter is not crucial for synthesis, it is highly advantageous. The solutions found by unification are necessarily unique (up to extensionality), and using synthesis to find such solutions would almost certainly be more computationally expensive.

We implement a syntax directed, first-order unification algorithm. Since evaluation outputs WHNF terms, anything inside lambda expressions is not normalized, and the unification normalizes any such term before equating its body with another expression. One reason for the lazy approach is exactly that unification can proceed iteratively. Two equated terms are reduced to WHNF, normalizing their topmost structure and potentially deferring a lot of computation. If the topmost structure matches, the computation is simply resumed for the unevaluated subexpressions and the process continues recursively. If, on the other hand, the topmost structure does not match at any point, equality check fails immediately, potentially *avoiding* a lot of computation.

The implementation is standard [McBride, 2003], and we only describe it informally. Throughout the execution, the algorithm keeps track of the current substitution, the yet unconsidered equations, and the unsolved equations. Starting with a list of equations to be unified, it repeatedly analyzes the head of the list, potentially inserting more equations to the front, until no equations remain, or a conflict is reached. If all equations are analyzed successfully, the algorithm terminates and returns the substitution and unsolved equations. Otherwise, a conflict is reached and the algorithm fails. At each step, there are four possibilities depending on the next equation under consideration:

1. Topmost constructs of the two sides of the equation match. In this case, new equations are pushed to be considered by the algorithm, equating the respective subexpressions.

2. Topmost constructs do not match. Then, either one side is a lambda term and the other not, and η-equality is checked; or unification fails.

3. At least one side of the equation is a meta variable. In this case, the algorithm performs an occurs-check. If it succeeds, the meta variable is solved with the value on the other side of the equation. Otherwise unification fails.

4. At least one side of the equation is an application of a meta variable to some arguments. This is a higher-order constraint that cannot be solved by our algorithm, and the equation is output as unsolved.

In the rest of this document, we use the notation $\sigma; \Delta \rightsquigarrow \sigma'; \Delta'$ to express that unifying the list of equations Δ under current substitution σ succeeds and produces a substitution σ' and a list of unsolved equations Δ'. Furthermore, we use the notation $s \stackrel{?}{=} t$ for a constraint equating terms s and t, and $\sigma[?_i \mapsto t]$ for the substitution obtained by adding a mapping of $?_i$ to t to a substitution σ

5.4 Type checking

Type checking largely follows the standard practice in dependent type systems. We implement a bidirectional algorithm inspired by the $\lambda\Pi$ [Löh et al., 2010], extending it with meta variables and the component type wrapper. A notable distinction compared to real-world implementations is the lack of distinction between the elaboration and the type checking phase. For simplicity of implementation, we merge the two phases. Holes are replaced by fresh meta variables as they are encountered during type checking, and the necessary bookkeeping for synthesis is done during type checking. Hence, our type checking additionally outputs a term resulting from the elaboration. To reflect this, we use the notation $\Gamma; M \vdash t :_{\downarrow} \tau \Rightarrow M'; \Delta; \sigma$ to express that checking a term t for type τ under a context Γ and existing metas M succeeds and produces the new set of metas M', unsolved equations Δ, and solutions to resolved metas σ. Analogously, we write $\Gamma; M \vdash t :_{\uparrow} \tau \Rightarrow M; \Delta; \sigma$ to state the same for inferring a term t has a type τ. Typing rules of our type system are shown in figure 5.3, where the initial letter of the label of each rule indicates whether the rule is checking or inferring the type, and the remainder of the label indicates the syntactic construct it relates to. The only exception is the CINF rule, where the type checker "changes direction"; if an inferrable term is being checked, the system infers the type, then attempts to unify it with the type it is checking against. For the sake of brevity, not all the typing rules are shown. We show the rules for non-standard constructs, and where we extend or modify them to fit the need of our synthesis framework. The omitted rules are essentially the same as the ones of λ_{Π}, extended with the additional input metacontext and the output metacontext, constraints and substitution. In the omitted rules with two type checking/inferring assumptions, the outputted information of the assumptions is threaded through and combined as in the ILET rule, left-to-right.

5.5 Component designator

A special component designator is introduced for the synthesis procedure to be well behaved in conjunction with datatype encoding. All data is encoded as functions, and our synthesis system always η-expands a goal with the function type. On the other hand, we typically want to provide some primitive operations for each given data type, and have the synthesizer construct programs using those primitive operations. The component designator $[\cdot]$ is intended to wrap a definition of a type for which the user provides some operations, such that the synthesis procedure

$$\frac{i = |M| + 1}{\Gamma; M \vdash \_ :_{\downarrow} \tau \Rightarrow M, (\Gamma, ?_i, \tau); \emptyset; \emptyset}\ \textsc{CHole} \qquad \frac{(\Gamma, ?_i, \tau) \in M}{\Gamma; M \vdash ?_i :_{\uparrow} \tau \Rightarrow M; \emptyset; \emptyset}\ \textsc{IMeta}$$

$$\frac{\Gamma; M \vdash s :_{\uparrow} \rho \Rightarrow M'; \Delta; \sigma \quad \Gamma, x : \rho; M' \vdash t :_{\uparrow} \tau \Rightarrow \mathcal{M}''; C'; \sigma'}{\Gamma; M \vdash x := s; t :_{\uparrow} \tau \Rightarrow \mathcal{M}''; C', \mathcal{C}''; \sigma}\ \textsc{ILet}$$

$$\frac{\Gamma; M \vdash t :_{\uparrow} \tau \Rightarrow M'; \Delta; \sigma \quad \sigma; \tau \stackrel{?}{=} \tau' \rightsquigarrow \sigma'; \Delta'}{\Gamma; M \vdash t :_{\downarrow} \tau' \Rightarrow M'; \Delta, \Delta'; \sigma'}\ \textsc{CInf}$$

Figure 5.3: The selected typing rules of $\mathcal{L}^{?}_{\Pi}$

knows not to eta-expand the goal with a component type, and uses the provided primitive operations instead. It is otherwise transparent to all other parts of the system, which treat any component-wrapped term $[t]$ the same as the unwrapped term t. In figure 5.4 we see an example of encoding the type of booleans, along with its two constructors true and false, and its recursor ite. The idea is that the component wrapper will instruct the synthesis system to treat Bool differently than $(A : \text{Type}) \to A \to A \to A$, reflecting the our intent of using it as an encoding of the boolean data type. If, on the other hand, we were to wrap a non-type term into the component designator, this would have no visible effect.

$$\begin{aligned}
&\text{Bool} = [(A : \text{Type}) \to A \to A \to A] : \text{Type};\\
&\text{true} = \lambda\ A\ x\ y\ .\ x : \text{Bool};\\
&\text{false} = \lambda\ A\ x\ y\ .\ y : \text{Bool};\\
&\text{ite} = \lambda\ A\ b\ x\ y\ .\ b\ A\ x\ y : (A : \text{Type}) \to \text{Bool} \to A \to A \to A;\\
&?_0 : \text{Bool} \to \text{Bool} \to \text{Bool}
\end{aligned}$$

Figure 5.4: Example of intended use of the component designator.

Chapter 6

Synthesis procedure

In this chapter, we present the main technical contribution of this thesis, the synthesis procedure. We begin with a high-level overview. Then we lay out some notation that we are going to use, followed by the definition itself. Finally, we discuss the non-essential aspects of the procedure which are left unspecified by the formal definition, but are relevant for implementation.

6.1 Overview

Procedure is framed in terms of iterative refinements of an incomplete program represented by the search state. The search state represents a term and unsolved constraints that must be satisfied for any program derived from the state to be well typed. Given an initial state represented by the user input, the system searches for a sequence of refinements that completes the program. The algorithm keeps a priority queue of states derived from the initial state, each representing a leaf of the search tree. The priority is determined by a heuristic which is itself a parameter. The algorithm then repeatedly selects the state at the top of the queue, selects a meta variable to be refined, and generates all feasible refinements according to its type. There are two cases for the refinement generation. In case the goal type is function, we η expand. Otherwise we iterate through the context and create a refinement for each variable that applies. For each refinement the algorithm runs a **propagation step**. This ensures that the refinement is consistent with the existing constraints on the refined state. Consequently, some constraints may become solvable and imply solutions to more meta variables. Conversely, some constraints may become violated, indicating that the current search branch is inconsistent and cannot lead to a well-typed solution. In the latter case the search procedure backtracks and avoids any unnecessary further search. The process repeats until either the selected state is complete, in which case the search succeeds, or the queue is empty, in which case the search fails.

6.2 Notation

Before we formally describe the system, we introduce some notation. The synthesis is defined in terms of the **refinement relation** $\longmapsto$, a binary relation on **search states**. A **search state** is a triple $(M; \Delta; \sigma)$. M is a collection of triples $(\Gamma, ?_i, \gamma)$ of meta variables created so far, with their typing context Γ and type γ. Δ is the collection of unsolved constraints, and σ is a substitution mapping solved meta variables to their solution. We will use the symbol Σ to denote an arbitrary search state when we have no need to refer to the individual fields of the triple. A sequence of zero or more refinements interspersed with constraint propagation is denoted by $\overset{*}{\longmapsto}$.
The overbar notation $\overline{ex_i}^{i<k}$ stands for the sequence of expressions ex_i for each $1 \leq i < k$. Similarly for $i \leq k$. For example, we write $\overline{(x_i : \tau_i)}^{i<3} \to \tau$ as a shorthand for $(x_1 : \tau_1)(x_2 : \tau_2) \to \tau$. The arrows between the domain abstractions are implicit, in line with the standard telescope notation.
$\text{Open}_\sigma(M)$ is the set of all triples in M whose meta variable is not mapped to a value by σ: all the open meta variables under σ.
The predicate FnType$(\cdot)$ stands for $\cdot = (x : \tau) \to \tau'$ for some fresh x, τ, τ'. Simply, it states that a term is a function type.
The predicate on search states Complete$(M; \Delta; \sigma)$ is true if and only if the search state has no open meta variables: $\text{Open}_\sigma(M) = \emptyset$.
Finally, we write $\vdash p \Uparrow p'$ to express that synthesis completes the program p, resulting in a complete program p'.

6.3 Formal system

The synthesis system is presented formally in 6.1, and we will now walk through each of the rules.

SYNTH

The entry point of the algorithm is the SYNTH rule, which states that in order for the synthesis problem to successfully complete the program $p_?$, we must successfully run type inference on $p_?$, obtaining the initial state Σ. This has ensures that both the initial incomplete program as well as the specification are well formed and the initial program does not violate the specification. Furthermore it inserts fresh meta variables in place of holes and does the necessary book keeping for the synthesis to proceed. Then, we must find a sequence of refinements from the initial state Σ to some complete state Σ'.

SSTART and SSTEP

Once all the preparatory work is done, we begin the synthesis. The "main loop" of the system is embodied by the SSTART and the SSTEP rules. SSTART states

that any state can reach itself, it is there simply as the base case of the inductive structure we are building. SSTEP then states that in order to extend a sequence of refinements, we must (1) provide a refinement of the final state of the sequence, and (2) successfully perform unification on the constraints emitted by the added refinement. The unification step after each new refinement propagates the local changes made by the refinement. This is step is crucial. It ensures that each new refinement added to the sequence preserves well-typing, supposing that constraints are eventually solved. Therefore, if unification fails, we know that adding the current refinement cannot lead to a solution, and the current search branch may be pruned. Otherwise, any emitted constraints are added to the next search state, and the search continues.

Creating refinements

There are two ways to create a valid refinement, represented by the rules SAPP and SLAM.

SAPP

If the next meta variable to be refined has a non-function type, then SAPP applies. This is the most involved rule and is where most of the work is done. The idea is to either solve the meta variable with a constant of the appropriate type or with a function applied to fresh meta variables for each of its arguments. Let us consider each case separately.

In the case that a constant is selected, we have $n = 0$ for the rule's variable. Then, all the sequences of expressions denoted by the overbar notation are empty, and no new meta variables are added. Hence the rule reduces to the following:

$$\frac{\begin{array}{cc}(\Gamma; ?_i; \gamma) \in M_1 & \neg\text{FnType}(\gamma) \\ f : \tau \in \Gamma & \neg\text{FnType}(\tau) \\ \sigma(\tau) \Downarrow \rho & \sigma; \rho \stackrel{?}{=} \gamma \rightsquigarrow \sigma'; \Delta' \\ \multicolumn{2}{c}{\sigma'' = \sigma'[?_i \mapsto f]}\end{array}}{M_1; \Delta; \sigma \longmapsto M_1; \Delta, \Delta'; \sigma''}$$

Meta $?_i : \gamma$ is picked to be solved with the constant $f : \tau$, where both γ and τ are *not* function types. Since meta variables may occur in τ and some may already be solved, we apply the current substitution to τ and evaluate it, calling the result ρ. Then, under current solutions σ, the types must be successfully unified, resulting in (potentially) extended solutions σ' and zero or more unsolved constraints Δ'. With unification succeeding, f is a valid candidate for solving $?_i$ and the substitution is extended with the mapping of $?_i$ to f. The old state is then refined by appending the new constraints to the existing ones, and replacing the old substitution with the extended one.

In the case that a function $f : \overline{(x_k : \tau_k)}^{k \leq n} \to \tau$ is selected to solve the meta variable, more work needs to be done, and the search will have to continue in order to construct the arguments to f. f is a function of n arguments, and a fresh meta variable must be created for each of them. The type of each meta variable must be the type of the corresponding formal parameter. However, due to dependent typing, the type of each argument can depend on the values of any previous arguments. To account for this, when we add the meta variable $?_{l+j}$ for the parameter $(x_j : \tau_j)$ to the state, we assign it the type $\tau_j \overline{[?_{l+k}/x_k]}^{k<j}$, substituting all the previous meta variables for their corresponding formal parameters. Next, in order for the refinement to be valid, the codomain of the function must be unified with the goal type. τ may contain meta variables, and may depend on the formal parameters of the function, which must be accounted for. It may happen that a solution of a meta variable contains formal parameters of the function, thus we first apply the current substitution $\sigma(\tau)$, and then substitute all the newly created meta variables for respective formal parameters $\sigma(\tau)\overline{[?_{l+k}/x_k]}^{k \leq n}$. The said substitutions may introduce redexes, so we normalize the obtained term and call the result ρ. Finally, we unify ρ with the goal type γ under the current substitution σ, obtaining new substitution σ' and additional constraints Δ. Supposing that the said unification succeeded, we solve selected meta variable $?_i$ with f applied to fresh meta variables $\overline{?_{l+k}}^{k \leq n}$.

Notice that, in case that the goal type is wrapped in a component designator, then SAPP rule applies regardless of whether the wrapped type is a function type or not. This is the reason why we introduce the construct.

SLAM

Finally, if the next meta variable has a function type, then the SLAM rule applies. In this case, we simply η-expand: the meta variable is solved by a lambda term whose body is a fresh meta variable. This does not reduce the set of programs we can construct up to $\beta\eta$-equivalence. We only enumerate β-normal terms, and any such solution that we do not enumerate has a $\beta\eta$-equivalent solution that we do. [Dowek, 1993]. This is a beneficial symmetry breaking criterion, as it avoids enumerating equivalent solutions.

6.4 Search strategies

The formal system presented in 6.3 does not fully specify a search algorithm. The presentation is purposefully left agnostic with respect to the search strategy. This is because the main contribution of this work, the idea of tracking and propagating constraints for early pruning, is independent of it. Nevertheless, a search algorithm must determine an order in which enumerates the candidate solutions, and the choice has a significant practical impact. We will now motivate the choices we made in our implementation.

$$\frac{(\Gamma, ?_i, (x:\tau) \to \tau') \in \text{Open}_\sigma(M) \qquad j = |M| + 1 \qquad \sigma' = \sigma[?_i \mapsto \lambda x.?_j] \qquad M' = M, ((\Gamma, x:\tau), ?_j, \tau')}{M; \Delta; \sigma \longmapsto M'; \Delta; \sigma'} \text{ SLAM}$$

$$\frac{\begin{array}{c}(\Gamma; ?_i; \gamma) \in \text{Open}_\sigma(M_1) \quad \neg\text{FnType}(\gamma) \quad l = |M_1| \\ f : \overline{(x_k : \tau_k)}^{k \le n} \to \tau \in \Gamma \quad \neg\text{FnType}(\tau) \\ M_{j+1} = M_j, (\Gamma, ?_{l+j}, \tau_j \overline{[?_{l+k}/x_k]}^{k<j}) \quad \text{for } 1 \le j \le n \\ \sigma(\tau)\overline{[?_{l+k}/x_k]}^{k \le n} \Downarrow \rho \qquad \sigma; \rho \stackrel{?}{=} \gamma \rightsquigarrow \sigma'; \Delta' \\ \sigma'' = \sigma'[?_i \mapsto f\ \overline{?_{l+k}}^{k \le n}]\end{array}}{M_1; \Delta; \sigma \longmapsto M_{n+1}; \Delta, \Delta'; \sigma''} \text{ SAPP}$$

$$\frac{}{\Sigma \stackrel{*}{\longmapsto} \Sigma} \text{ SSTART} \qquad \frac{\begin{array}{c}\Sigma_1 \stackrel{*}{\longmapsto} \Sigma_2 \\ \Sigma_2 \longmapsto M_3; \Delta_3; \sigma_3 \\ \sigma_3; \Delta_3 \rightsquigarrow \sigma_4; \Delta_4\end{array}}{\Sigma_1 \stackrel{*}{\longmapsto} M_3; \Delta_4; \sigma_4} \text{ SSTEP}$$

$$\frac{\vdash p_? :\uparrow \tau \Rightarrow \Sigma \qquad \Sigma \stackrel{*}{\longmapsto} M; \Delta; \sigma \qquad \text{Complete}(M; \Delta; \sigma)}{\vdash p_? \Uparrow \sigma(p)} \text{ SYNTH}$$

Figure 6.1: Synthesis rules for $\mathcal{L}^?_{\text{II}}$

State selection

As explained in 6.1, the states are kept in a priority queue and a user-provided heuristic is used to assign priorities. In our evaluation we used a heuristic that prioritizes smaller programs with fewer unsolved meta variables, breaking the ties according to the FIFO principle. Intuitively, we perform a breadth-first search, biased by the expected size of the solution.

Meta variable selection

Given an incomplete program under the search state $(M; \Delta; \sigma)$, for the search to progress, we must provide a procedure for selecting the next meta variable for refinement. In the previous work on synthesis in dependent type theory it has been observed that selecting meta variables right to left is beneficial because it can significantly reduce the search [Dowek, 1993]. The reason behind this is the fact that the type of a subterm can depend on the terms to its left, so solving the meta variable on the right can imply the solution of the meta variabless its type depends on. For example, if we have $?_i : \mathbb{N}$ and $?_j : P\ ?_i$, then instantiating $?_j$ to some value $v : P\ n$ implies the solution of $?_i$ to be n. If $?_i$ was solved first, we would potentially waste a lot of time searching for solutions with infeasible values, since there may be many possible assignments for $?_i$ and any meta variables between it and $?_j$.

Variable selection

If the SAPP rule applies, we must specify an order in which to enumerate candidate variables from the context. We simply enumerate the candidates right to left; from the most recently introduced variable backward. This is motivated by the intuition that more recently introduced variables are more likely to be used. Nevertheless, with the state selection strategy we are using, this choice has fairly little impact.

Chapter 7

Evaluation methods

In this chapter we describe and motivate our evaluation methods. First, we state and motivate the objective of the evaluation. Next, we explain and motivate the choice of experiments. Finally, we describe the setup we used to evaluate.

7.1 Objective

There are two main objectives for the evaluation: (1) assessing the impact of constraint propagation to synthesis, and (2) comparing our system to state-of-the-art.

Impact of propagation Our system runs unification on all unsolved constraints after each refinement. While we may spend more computational effort per search state, this could lead to dramatic decrease in the number of search states explored. We intend to assess whether this pays off (**RQ1**). To do this, we are going to measure both the number of states explored (**RQ1.1**), and the total time spent (**RQ1.2**) on various synthesis problems. We are going to perform these measurements with the naive approach and with our constraint-guided approach. For the naive approach we chose the one that collects and defers unification of all unsolved constraints until reaching complete programs. We deem this the more appropriate choice since, in theory, the two systems can synthesize the same set of programs, but we expect one to do so more more efficiently.

Comparison with state-of-the-art The overarching goal of this research is to explore the idea of using dependent type theory as a basis for a program synthesis framework. One major benefit of a dependently typed framework is the fact that it can naturally express a broad range of specifications. Thus, one question we want to answer is whether, and how easily, our system can express the forms of specification accepted by current state-of-the-art (**RQ2**). On the other hand, performance is a crucial aspect of synthesis systems. While our system is a small prototype with little optimization, it is still interesting to compare its performance to that of leading-edge systems (**RQ3**).

7.2 Choice of experiments

We now motivate the choice of our experiments. Many problems were ported from previous work for comparison. Some were more heavily adapted, and the rest were created by the author.

We chose two systems for comparison with previous work: MYTH and SYNQUID. The said works both provide a solution for synthesis of total functional programs, instead of targeting imperative programs or a domain specific language. However, the main reason we chose these two is the form specification they accept. MYTH accepts an IO example-based specification and simple, monomorphic types, while SYNQUID uses polymorphic refinement types; a highly expressive, logic based specification. Moreover, their specification forms are highly disjoint: MYTH cannot express any polymorphism nor refinements, while SYNQUID can not express any IO examples. Since this thesis proposes a unifying framework for program synthesis, with a goal of allowing broad and flexible forms of specification, we find the choice for comparison appropriate.

The problems created by the author fall into two categories. In one category are problems that only use a polymorphic type to express the specification. The other are problems that use a mixed specification, an important feature of the proposed approach.

7.3 Setup

We describe the setup for the experiments. First, the show the differences between the naive and the constraint-guided system. Next, we discuss how we ported the experiments from past work. The experiments and the script used to run them can be found in the project repository.

7.3.1 Naive system

The naive system differs from the constraint-guided one in two rules, namely SSTEP and SYNTH. Instead of propagating the constraints after each refinement in SStep, we simply collect them and thread them through. This amounts to removing the propagation step:

$$\frac{\Sigma_1 \overset{*}{\longmapsto} \Sigma_2 \qquad \Sigma_2 \longmapsto \Sigma_3}{\Sigma_1 \overset{*}{\longmapsto} \Sigma_3}\ \text{SSTEP}^\dagger$$

Now, any additional constraints generated in SAPP are simply added to the existing ones; SAPP already does this. SLAM does not change the constraints.

Next, we must ensure that once a complete state is reached, we only accept it as a solution if all the constraints can be satisfied. Thus, we modify the SYNTH rule

to require that all the collected constraints can be solved:

$$\frac{\vdash p_? :_\uparrow \tau \Rightarrow \Sigma \qquad \Sigma \overset{*}{\longmapsto} M; \Delta; \sigma \qquad \text{Complete}(\Sigma') \qquad \sigma; \Delta \rightsquigarrow \sigma'; \emptyset}{\vdash p_? \Uparrow \sigma'(p)} \text{SYNTH}^\dagger$$

7.3.2 Adapting experiments

The ported experiments are adapted to $\mathcal{L}^?_\Pi$. This involves translating the specification into a type, encoding the relevant data types, their recursors and their constructors. In many problems, not every recursor and constructor of every data type is needed for the solution. However, we deemed it fair to always include all the constructors and the recursor for each encoded data type. Leaving out any of the said definitions assures that our system will never attempt to use it, while in the languages with built in algebraic data types, all such definitions are available wherever the data type is in scope. Thus leaving out any definitions we know should not be used gives our system an unfair advantage.

When encoding example-based specification, the proofs by reflexivity were given to inform the system that the two sides of the equations must compute to the same value. This is nothing more than stating that the said equations are IO examples.

While refinement types are naturally expressible in a dependent type theory, due to a limitation of our language we can not express *programs* that operate on refinement types. However, some of SYNQUID's benchmarks can be re-interpreted in terms of dependent types. Namely, functions operating on polymorphic list types refined by a length specifying predicate can be expressed as functions on Vec – the family of fixed length lists. There is a distinct set of experiments containing these problems.

Chapter 8

Results

In this chapter we present the results[1] of the research conducted for this thesis. We present the performance of our system with and without the propagation step on various benchmarks. The last four columns of the table are interpreted as follows: prefix #S- stands for number of states explored until the solution was found, and T- for time it took to synthesize. Suffix -CG stands for constraint-guided, and -N for naive search. Time was measured in seconds and rounded to two decimals. n/a means that the system did not find any solution within 2 minutes. The problems are divided into categories. Example based problems are divided into four categories depending on the primary data type manipulated by the synthesized function: booleans, natural numbers, (monomorphic) lists, and (monomorphic) binary trees. The category labeled "logical" is the category of re-interpreted SYNQUID problems.

Table 8.1: Experimental results

Test name	#S-CG	#S-N	T-CG	T-N
Examples bool				
examples_bool_and	16	27	0.1	0.09
examples_bool_or	16	27	0.08	0.08
examples_bool_implication	42	97	0.39	0.31
examples_bool_negation	27	53	0.13	0.11
examples_bool_xor	776	1967	10.42	9.02
Examples nat				
examples_nat_is_even	935	1514	14.78	17.16
examples_nat_add	41	207	1.13	3.19

[1]The experimental setup, along with a script for running the experiments can be found on the following repository: `https://github.com/Herb-AI/HerbDependentTypes.jl`

Test name	#S-CG	#S-N	T-CG	T-N
examples_nat_max	729	1874	26.23	48.06
examples_nat_pred	1013	n/a	14.59	n/a
Examples list				
examples_list_append	219	1660	13.56	73.24
examples_list_compress	n/a	n/a	n/a	n/a
examples_list_concat	301	618	10.79	16.83
examples_list_drop	163	924	13.82	60.97
examples_list_even_parity	n/a	n/a	n/a	n/a
examples_list_map	678	n/a	37.23	n/a
examples_list_take	n/a	n/a	n/a	n/a
Examples tree				
examples_tree_count_nodes	n/a	n/a	n/a	n/a
examples_tree_inorder	n/a	n/a	n/a	n/a
examples_tree_map	644	1471	35.18	67.7
examples_tree_preorder	n/a	n/a	n/a	n/a
Polymorphic				
polymorphic_identity	3	3	0.04	0.0
polymorphic_return_second	5	5	0.0	0.0
polymorphic_pair_proj1	8	8	0.0	0.0
polymorphic_S_combinator	10	10	0.01	0.0
polymorphic_pair_snd_and_fst	98	98	0.07	0.07
polymorphic_nonempty_list_get	24	24	0.08	0.03
Logical				
vec_cons	n/a	n/a	n/a	n/a
vec_append	n/a	n/a	n/a	n/a
vec_stutter	n/a	n/a	n/a	n/a
vec_self_append	n/a	n/a	n/a	n/a
vec_map	n/a	n/a	n/a	n/a
vec_tree_to_vec	n/a	n/a	n/a	n/a
Mixed				
examples_and_partial_nat_pred	646	1429	16.42	39.59

Test name	#S-CG	#S-N	T-CG	T-N
polymorphic_examples_list_map	n/a	n/a	n/a	n/a
partial_vec_append	195	301	1.35	2.27
examples_vec_append	n/a	n/a	n/a	n/a
examples_vec_reverse	n/a	n/a	n/a	n/a
examples_and_partial_vec_reverse	154	n/a	16.67	n/a
partial_vec_stutter	n/a	n/a	n/a	n/a
examples_and_partial_vec_stutter	n/a	n/a	n/a	n/a
partial_vec_self_append	140	197	1.31	2.49
partial_tree_to_vec	n/a	n/a	n/a	n/a

Chapter 9

Discussion

This chapter discusses the results laid out in Chapter 8. First, we discuss the impact of incorporating constraints into the search, answering the first research question. Next, we discuss the capabilities of our system in comparison with two other previous works, answering the second and the third research question. Finally, we acknowledge and discuss the limitations of our system.

9.1 Impact of propagation

RQ1 is answered positively. We compare the performance of the naive and the constraint-guided approach, to establish the impact of propagation on. Overall, the constraint-guided significantly outperforms the naive one. First, let us address the exceptions.

Observe that for the polymorphic problems both methods explore the same number of states. This is because all the constraints can be solved immediately, so there is no potential for exploiting unsolved constraints. Intuitively, their types impose less restriction relative to indexed data types. Here, we see that both algorithms perform similarly. This is in line with our expectation, since there are no intermediate constraints to propagate and hence no additional work for the constraint-guided algorithm.

Furthermore, on some of the simple example-based problems for creating boolean functions, we see that, despite the reduction in explored states, naive approach is somewhat faster. This, too, is to be expected since, for such a small problem, the benefit of pruning does not outweigh the relatively high cost of propagation.

However, on all the other problems, the constraint-guided approach performs significantly better, both in terms of states explored (**RQ1.1**) and time it takes to find a solution (**RQ1.2**). This strongly indicates that propagation *pays off* in the general case. While we spend more time per state, the pruning of the search space that it enables significantly outweighs the additional cost of propagation in all but the smallest problems.

9.2 Comparison with other systems

Based on the results presented in 8, we compare our system with MYTH and SYNQUID, in terms of expressivity of specification and performance.

9.2.1 Range of specification

RQ2 is answered positively. We demonstrate that, thanks to dependent types, our system can express a much broader range of specification than either of the compared systems. This includes IO examples, refinement types, partial programs, indexed data types such as vectors, and any combination of those. Expressing IO examples is somewhat more verbose than in MYTH, which provides dedicated syntax for it. Nevertheless, is easy and intuitive, since it closely reflects familiar mathematical notation. Expressing refinement types is immediate, as they can be seen as a special case of dependent sum types. Combining different forms of specification is straightforward too.

9.2.2 Performance

RQ3 is answered negatively. Performance of our system is significantly worse then the previous work on their respective domains. This is expected since our system is not tailored to any specific form of specification while covering a very wide range. We did not perform any measurements on the compared system ourselves, hence we do not present any data about them. Nonetheless, the respective papers [Osera and Zdancewic, 2015] [Polikarpova et al., 2016] present measurements done by the authors, where we see that all the problems that we compare against are solved within one second. Some surprising findings are that none of our re-statements of SYNQUID's problems in terms of indexed types were solved successfully. The reason behind this is the limitation of our system to first order unification. Defining required programs requires using the recursor of Vec, whose motive is of type Nat $\rightarrow$ Set. Failing to establish it through unification causes the search to devolve to naive search since we lose any certainty about typing of the arguments to the recursor. Furthermore, instead of helping the situation, combining indexed types and examples significantly slows down the time needed to process a single search state, ultimately defeating the purpose. We suspect the reason for this is the fact that many constraints remain unsolved and accumulate, while after each refinement the system runs unification on all the constraints, spending a lot of time for no benefit. Lastly, we see that the system fails to solve a seemingly very simple problem: predecessor of a natural number. The reason behind this is the fact that the definition of this function for encoded numbers is surprisingly subtle.

9.3 Limitations

While we demonstrate significant improvement over the naive approach, our implementation has several major limitations. We now address these.

Recursors The only way for us to express recursive functions is through the use of recursors. This requires explicitly stating the motive – the intended codomain – whenever we want to recurse on a data structure. In some cases the motive can be inferred automatically by unification, however, in other cases it will succeed with unsolved constraints. This will happen whenever all first-order constraints are solvable, but some higher-order ones occur too. For example consider synthesizing the map function for Vec. Its type is $(n : \mathbb{N})(A : \text{Type})(B : \text{Type}) \to (A \to B) \to (\text{Vec } n\ A) \to (\text{Vec } n\ B)$. Suppose we are refining $\lambda\ n\ A\ B\ f\ v\ .\ ?_0$ by SAPP into the following partial program:

$$\lambda\ n\ A\ B\ f\ v\ .\ \text{foldVec}\ ?_1\ ?_2\ ?_3\ ?_4\ ?_5\ ?_6$$

Recall that the type of Vec's recursor is

$$\begin{aligned}(A : \text{Type})(n : \mathbb{N})(P : \mathbb{N} \to \text{Type})\\ \to P\ 0 \to ((m : \mathbb{N}) \to A \to P\ m \to P\ (\text{succ}\ m))\\ \to \text{Vec}\ A\ n \to P\ n.\end{aligned}$$

Thus, the type of the body of our partial program is $(P\ n)[?_2/n][?_3/P] = (?_3\ ?_2)$. On the other hand, the required type is Vec $n\ B$. However, the unification problem $(?_3\ ?_2) \stackrel{?}{=} \text{Vec}\ n\ B$ is not first order, and it is simply emitted as unsolved. Now, the types of our meta variables $?_4$ through $?_6$ are $(?_3\ 0)$, $(m : \mathbb{N}) \to A \to ?_3\ m \to ?_3\ (\text{succ}\ m)$ and $(?_3\ n)$, respectively. All three involve meta variables applied to other terms, and will hence themselves not be first-order solvable. This has an unfortunate consequence that any type will conditionally unify with these, and hence for any variable in the context we will create a refinement. Thereby, we have devolved to naive search.

Another difficulty of recursors is that they do not allow for simple definitions of certain conceptually simple functions. A good example is the predecessor function $\text{pred} : \mathbb{N} \to \mathbb{N}$. With recursors, we have the following, surprisingly involved definition:

$$\lambda\ n\ A\ f\ x\ .\ n\ ((A \to A) \to A)\ (\lambda\ g\ h\ .\ h\ (g\ f))\ (\lambda\ u\ .\ x)\ (\lambda\ u\ .\ u)$$

Induction A major limitation of encodings is the lack of induction principle. Without it, our capability of defining functions on indexed types is limited. This is why we can not solve direct translations of SYNQUID's benchmarks. For example, even defining a simple function such as $\text{double} : \mathbb{N} \to \Sigma\ \mathbb{N}\ (\lambda\ n\ .\ \text{Even}\ n)$, requires the induction principle on the natural numbers.

Ergonomics The usability of our system is severely reduced because of the reliance on encodings. A lot of overhead is introduced both syntactically and intellectually.

None of the limitations discussed so far are inherent to constraint-driven synthesis in a dependently typed framework. Rather they are induced by our reliance on data type encoding. Nevertheless, there is one limitation that is more fundamental to the approach itself.

Generality While generality of specification of our system is an attractive aspect, it comes at a cost. Because such a wide range of specification can be expressed, the system must refrain from making too many assumptions that could then be exploited in the design. In comparison, both MYTH and SYNQUID take advantage of the limited form of specification they allow. MYTH uses a custom data structure designed to efficiently propagate examples to subproblems generated by refinements. Analogously, SYNQUID uses a specialized solver to efficiently compute refinements for sub problems. On the other hand, our approach uses the same mechanism for solving problems regardless of their specification.

Chapter 10

Future work and conclusion

10.1 Future work

We laid out several limitations of the system in 9.3. In each of them we see opportunities for improvement of constraint-guided synthesis in a dependently typed framework.

Higher-order unification Implementing a more powerful unification algorithm could drastically improve the performance of the system. Recall from 9.3 that when using functions on indexed data types, we run into the problem of solving higher-order constraints. These often fall into decidable fragments of higher-order unification such as pattern fragment [Miller, 1992]. Moreover, type checkers of modern dependently typed languages such as Agda or Lean all include such algorithms [Norell, 2007][de Moura et al., 2015]. Thus, we believe that it would be highly beneficial for the performance of our system if it was augmented with more powerful unification.

Inductive data types Modern dependently typed languages all provide builtin indexed inductive data types. Two major benefits that they would bring to our system are presence of the induction principle and ergonomics. On the one hand, the induction principle would allow us to prove a significantly richer range of specification than we now can. Recall that many problems from SYNQUID are simply not solvable in our current language due to the lack of induction principle, despite the fact that their specification is expressible in our type system. On the other hand, the difference in ergonomics provided by the builtin data types would be significant. Manually encoding inductive data types is tricky and requires considerably more effort than expressing them with builtin mechanisms of a typical dependently typed language. Thus, combining builtin data types with constraint-guided search has the potential to yield a much more expressive and user friendly synthesis system.

Heuristics We put little effort into search heuristics. As is, our algorithm roughly enumerates terms in the order of size. On the other hand, while Agsy only performs a fairly naive search, it is able to solve some interesting problems, largely due to carefully crafted heuristics. Furthermore, taking into account the unsolved constraints when selecting the next state and the next metavariable to refine seems like a particularly interesting research direction. The importance of the selection heuristics is well known in constraint satisfaction problems research [Mouhoub and Jafari, 2011].

Integration with solvers While we propose an internal approach to synthesis, we believe that using external solvers is largely orthogonal to our approach. Moreover, we consider that integrating external solvers *along* with our approach is an interesting direction for future research: specialized solvers could be used to enhance the synthesis by solving subproblems well-suited to their specific capabilities.

Since we can express such a wide range of type-based specification, we can envision a unifying framework for type-driven program synthesis methods based on a dependently typed language. With a more powerful unification algorithm, modern language features, an interface to external solvers and metaprogramming, the framework could operate along the following lines:

- users define custom data types to represent various forms of specification
- some data types represent problems solvable by a solver; for example, a properly restricted form of dependent sum type for representing refinement types
- using metaprogramming the user defines how the said custom type should be interpreted by a specialized solver and how the result should be reinterpreted back into the language, in the spirit of Schmitty [Kokke]
- the dependent type system ensures the consistency of solutions provided by different solvers via constraints

10.2 Conclusion

We implemented a dependent type-and-constraint driven synthesis system. Because of the dependent type driven approach, we can easily express and combine various forms of specification, providing great flexibility. We demonstrate that the constraint driven approach greatly improves naive search, significantly pruning the search space even with a first-order constraint solver. Nevertheless, due to the simplicity of our implementation and the breadth of specification, it is not competitive with state-of-the-art in terms of performance, which tends to target narrower forms of specification.

However, we see a lot of potential for future research. We identify a number of different research directions, and outline a hypothetical framework unifying type-driven solutions in a dependently typed system.

Bibliography

L. Augustsson. Djinn, 2014. URL `https://hackage.haskell.org/package/djinn-lib`.

H. P. Barendregt et al. *The lambda calculus*, volume 3. North-Holland Amsterdam, 1984.

A. Church and J. B. Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 39(3):472–482, 1936.

H. B. Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences*, 20(11):584–590, 1934.

L. de Moura, J. Avigad, S. Kong, and C. Roux. Elaboration in dependent type theory. *arXiv preprint arXiv:1505.04324*, 2015.

G. Dowek. A complete proof synthesis method for the cube of type systems. *Journal of Logic and Computation*, 3(3):287–315, 1993.

S. Foster and G. Struth. Integrating an automated theorem prover into agda. In *NASA Formal Methods Symposium*, pages 116–130. Springer, 2011.

H. Geuvers. Induction is not derivable in second order dependent type theory. In *International Conference on Typed Lambda Calculi and Applications*, pages 166–181. Springer, 2001.

S. Gulwani, O. Polozov, R. Singh, et al. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017.

R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, 146:29–60, 1969.

W. A. Howard et al. The formulae-as-types notion of construction. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, 44:479–490, 1980.

D. J. Howe. Equality in lazy computation systems. In *LICS*, volume 89, pages 198–203. Citeseer, 1989.

E. Kneuss, I. Kuraj, V. Kuncak, and P. Suter. Synthesis modulo recursive functions. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, pages 407–426, 2013.

P. Kokke and W. Swierstra. Auto in agda: programming proof search using reflection. In *Mathematics of Program Construction: 12th International Conference, MPC 2015, Königswinter, Germany, June 29–July 1, 2015. Proceedings 12*, pages 276–301. Springer, 2015.

W. Kokke. Schmitty: Agda bindings to smt-lib2 compatible solvers. URL `https://github.com/wenkokke/schmitty`.

G. W. Leibniz. *Discourse on Metaphysics*. Springer, 1686.

F. Lindblad and M. Benke. A tool for automated theorem proving in agda. In *International Workshop on Types for Proofs and Programs*, pages 154–169. Springer, 2004.

A. Löh, C. McBride, and W. Swierstra. A tutorial implementation of a dependently typed lambda calculus. *Fundamenta informaticae*, 102(2):177–207, 2010.

C. McBride. First-order unification by structural recursion. *Journal of functional programming*, 13(6):1061–1075, 2003.

D. Miller. Unification under a mixed prefix. *Journal of symbolic computation*, 14 (4):321–358, 1992.

R. Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

M. Mouhoub and B. Jafari. Heuristic techniques for variable and value ordering in csps. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 457–464, 2011.

U. Norell. *Towards a practical programming language based on dependent type theory*, volume 32. Chalmers University of Technology, 2007.

P.-M. Osera and S. Zdancewic. Type-and-example-directed program synthesis. *ACM SIGPLAN Notices*, 50(6):619–630, 2015.

N. Polikarpova, I. Kuraj, and A. Solar-Lezama. Program synthesis from polymorphic refinement types. *ACM SIGPLAN Notices*, 51(6):522–538, 2016.

R. Pollack. Implicit syntax. In *Informal Proceedings of First Workshop on Logical Frameworks, Antibes*, volume 4, page 66. Citeseer, 1990.

P. M. Rondon, M. Kawaguci, and R. Jhala. Liquid types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 159–169, 2008.

L. Skystedt. A new synthesis tool for agda. 2022.

A. Solar-Lezama. *Program synthesis by sketching*. University of California, Berkeley, 2008.