PROGRAM SYNTHESIS WITH TYPES

Peter-Michael Osera

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania
in
Partial Fulfillment of the Requirements for the
Degree of Doctor of Philosophy

2015

Supervisor of Dissertation

_____

Steve Zdancewic
Professor of CIS

Graduate Group Chairperson

_____

Lyle Ungar
Professor of CIS

Dissertation Committee

Stephanie Weirich (Professor of CIS, Committee Chair)

Benjamin Pierce (Professor of CIS)

Rajeev Alur (Professor of CIS)

Sumit Gulwani (Principal Researcher, Microsoft Corporation)

PROGRAM SYNTHESIS WITH TYPES

To Victoria, Talia, Felicity, Eliana, and Selena.

Acknowledgements

A dissertation is a long journey, but it is one not treaded alone (contrary to what the front page of this document says). I am greatly indebted to every person that has helped me along the way. Every professor, student, software developer, and colleague that I have interacted with has taught, guided, and challenged me in some way to become a better computer scientist. Without their assistance, this dissertation would have never seen the light of day.

However, several people deserve special mention. First, my deepest thanks go out to my advisor, Steve Zdancewic, for his guidance and insight over the last seven years. In addition to giving me the freedom to pursue whatever research I found interesting, he has been an excellent mentor and friend throughout this whole process. Whether it was a paper deadline, the academic job market, scheming new ways to improve the undergraduate curriculum, or identifying the best TV shows for our young daughters to watch, he has always been there to help me out whenever I needed it.

I also thank the remaining professors that make up the PLClub research group—Stephanie Weirich and Benjamin Pierce. In addition to the advice and encouragement that they have given me, they are responsible for maintaining the excellent quality of life within the group. Starting graduate school with a newborn in tow was an absolutely terrifying prospect, one that I would not have managed if the programming languages research group at Penn was not as warm, inviting, and caring (on top of being rigorous and hard-working) as it has been during my stay.

Related, I thank all the graduate students, post-docs, and undergraduates that have been a part of PLClub while I was at Penn, including: Nate Foster, Jeff Vaughan, Aaron Bohannon, Karl Mazurak, Jianzhou Zhao, Michael Greenberg, Chris Casinghino, Richard Eisenberg, Hongbo (Bob) Zhang, Arthur Azevedo de Amorim, Justin Hsu, Leonidas Lambropoulos, Jennifer Paykin, Robert Rand, and Antal Spector-Zabusky.[1] You have all been a constant source of information, inspiration, and motivation. I also especially thank the members of my research cohort—Daniel Wagner, Brent Yorgey, and Vilhelm Sjöberg—for being amazing human beings and friends. We made it, guys!

I have been honored to collaborate with some amazing people whose feedback

[1] And apologies if I missed your name, and we crossed paths in PLClub. A lot of people have come and gone during my tenure!

and input were instrumental in making this particular line of work on program synthesis as high quality as it could be. Thank you to David Walker, Jonathan Frankle, and Rohan Shah for all the hours of brainstorming, idea-batting, and work you have put into this project.

Even though this dissertation is about program synthesis with types, I consider myself a computer science educator as much as I consider myself a programming languages theorist. Thank you to Chris Murphy, Swapneel Sheth, Arvind Bhusnurmath, Benedict Brown, Katie Gibson, and Lili Dworkin, among others, for being my CS education outlet at Penn.

I thank my family for their love and support throughout this process. The road to a dissertation is every bit emotional as it is intellectual, and I could not have kept it together without them. Thank you to my wife and dearest friend, Victoria, for moving out with me to the East coast—away from all the friends and family we have ever known—to pursue this dream. And thank you to my kids—Talia, Felicity, and soon-to-be two more—for your love, patience, and understanding while daddy was away at work so long to finish writing his book. This dissertation, as well as everything else I ever do, is dedicated to all of you.

Finally, my work at Penn has been supported by a pair of grants that I would like to acknowledge:

- NSF Award CCF-1138996—Expeditions in Computer Augmented Program Engineering: ExCAPE: Harnessing Synthesis for Software Design.
- ONR award N000141110596—IRONCLAD C/C++: Enforcing Memory Safety to Prevent Low-level Security Vulnerabilities.

ABSTRACT

PROGRAM SYNTHESIS WITH TYPES

Peter-Michael Osera

Steve Zdancewic

Program synthesis, the automatic generation of programs from specification, promises to fundamentally change the way that we build software. By using synthesis tools, we can greatly speed up the time it takes to build complex software artifacts as well as construct programs that are automatically correct by virtue of the synthesis process. Studied since the 70s, researchers have applied techniques from many different sub-fields of computer science to solve the program synthesis problem in a variety of domains and contexts. However, one domain that has been less explored than others is the domain of typed, functional programs. This is unfortunate because programs in richly-typed languages like OCaml and Haskell are known for "writing themselves" once the programmer gets the types correct. In light of this observation, can we use type theory to build more expressive and efficient type-directed synthesis systems for this domain of programs? This dissertation answers this question in the affirmative by building novel type-theoretic foundations for program synthesis. By using type theory as the basis of study for program synthesis, we are able to build core synthesis calculi for typed, functional programs, analyze the calculi's meta-theoretic properties, and extend these calculi to handle increasingly richer types and language features. In addition to these foundations, we also present an implementation of these synthesis systems, Myth, that demonstrates the effectiveness of program synthesis with types on real-world code.

Contents

List of Figures

Chapter 1

Introduction

Type systems are the most ubiquitous form of formal verification tool present in programming languages today. They provide a number of important benefits for programmers, for example, static checking of errors and opportunities for optimizations. However, one of the most often overlooked benefits is that type systems also help programmers design programs more efficiently. In particular, typed, functional programming languages like ML and Haskell provide a compelling combination of rich types coupled with a succinct, yet powerful set of languages features. Programmers that use these languages frequently comment that once they figure out the types of their program, the program just writes itself!

Let's investigate this point in more detail. Consider writing a simple program, say the map function over lists, in a ML-like language. First, let's establish the type of the overall function: map is a higher-order function that takes (1) a function that transforms a single element of a list into some other type, (2) a list of the first type, and produces a list of the second type. This means that map has type $('a \to 'b) \to 'a\,\mathsf{list} \to 'b\,\mathsf{list}$ where $'a\,\mathsf{list}$ is ML's way of writing down a *polymorphic* or *generic* list. Here, $'a$ is a type variable that represents the type of elements of that list, its *carrier type*. Now, let's develop this function incrementally in a *type-directed* manner by keeping close track of the types of expressions we need to fill in at each point of the program. Initially, we have the following goal,

$$\blacksquare : ('a \to 'b) \to 'a\,\mathsf{list} \to 'b\,\mathsf{list},$$

where $\blacksquare$ represents our *goal*—the hole in the program we need to fill with an expression. Because the hole is at arrow type, it makes sense to introduce a function:

$$\mathsf{let\ rec\ map}\ (f{:}'a \to 'b)(l{:}'a\,\mathsf{list}) : 'b\,\mathsf{list} = \blacksquare : 'b\,\mathsf{list}.$$

With the function written down, our goal is to now fill in the body of the function which, by the definition of the function, must produce a $'b\,\mathsf{list}$. Note how the rich types greatly constrain the set of programs we can write at this point. Because the carrier type of the output list is unknown, we have no way of

constructing a non-empty $'b$ list. The only other value we can provide is the empty list which means map would produce the empty list for any pair of inputs which is not correct. We need more information to make progress, and we obtain that information by *pattern matching*:

$$\begin{array}{l}\mathsf{let\ rec\ map}\ (f{:}'a \to 'b)(l{:}'a\ \mathsf{list}) : 'b\ \mathsf{list} = \\ \quad \mathsf{match}\ l\ \mathsf{with} \\ \quad \mid\ \mathsf{Nil} \to \blacksquare : 'b\ \mathsf{list} \\ \quad \mid\ \mathsf{Cons}(x, l') \to \blacksquare : 'b\ \mathsf{list}.\end{array}$$

By pattern matching, we perform case analysis on some value. In this case, we know that a list in ML has two possible *constructors* or ways of making a value of type list: Nil representing the empty list and Cons representing the list composed of a single element x followed by the rest of the list l'. We are left with two goal expressions to fill in, both of type $'b$ list, corresponding to the branches of the pattern match.

In the Nil branch, we know the input l is the empty list at this point in the program, so it is sensible to produce the empty list. In the Cons branch, we know that the list has at least one element x of type $'a$ list and we have a handle on the rest of the list l'. With x, we now have a way of manufacturing a $'b$ list: applying x to f. With this, we can transform the head of list l and the cons that onto the result of recursively mapping over the tail of l. Thus, the final program is:

$$\begin{array}{l}\mathsf{let\ rec\ length}\ (l{:}\mathsf{list}) : \mathsf{nat} = \\ \quad \mathsf{match}\ l\ \mathsf{with} \\ \quad \mid\ \mathsf{Nil} \to \mathsf{Nil} \\ \quad \mid\ \mathsf{Cons}(x, l') \to \mathsf{Cons}(f\ x, \mathsf{map}\ l').\end{array}$$

At the end, we needed some ingenuity to recognize how to break down the mapping operation over a list in terms of its components. However, we can see how types guided our development process by constraining the allowable set of programs at different points of the program. The question is simple: can we automate this sort of type-directed reasoning? Given a type, and perhaps some additional specification of how a program should behave, can we derive a program of the appropriate type that meets this specification?

This process of generating programs automatically from specification is called *program synthesis* and is one of the greatest, longest-standing pursuits of computer science. At its core, program synthesis is a problem of combining *search* with *specification*. The domain of the search is the infinite sea of possible programs (for some particular programming language). Specification allows us to pick out particular programs of interest that we find during search. This specification can take many forms, for example, logical statements [Green, 1969; Manna and Waldinger, 1979], input/output examples [Albarghouthi et al., 2013; Feser et al.,

2015; Kitzelmann, 2010b; Summers, 1976], and partial programs [Alur et al., 2013; Singh et al., 2013; Solar-Lezama, 2008], among others. Regardless of the mode of specification we choose, the search and specification components of synthesis frequently inform each other; specification helps refine the space of possible programs, making search tractable, and different search techniques are more amendable to particular kinds of specification.

As you can imagine, program synthesis is a very difficult problem—we are still writing programs by hand, after all—and undecidable in the general case. However, it is worth pursuing because the benefits of program synthesis technology are immense. By synthesizing a program from a specification, we guarantee that the program agrees with that specification by virtue of the synthesis process. If that specification includes properties such as correctness or safety, then the program will enjoy those properties automatically. Furthermore, it is often easier to write down a specification of how a program should work than to write the program itself, especially if that specification is partial, for example, a collection of input/output examples or a demonstration of how the program should work. Program synthesis then becomes a tool that makes the power of computer programs more accessible to people, especially non-programmers, who can signify their intent, but do not know how to translate that intent into a program. Finally, because computer programs are general-purpose, synthesizing programs means synthesizing methods of solving any task we can express as a program. This might mean deriving a complex program to determine the trajectory of a rocket or simply automating the task of entering data into a spreadsheet [Gulwani, 2011].

In this work, we develop a *type-theoretic interpretation* of program synthesis. Type theory [Martin-Löf, 1984] provides a constructive foundation for all of mathematics. Programming language researchers leverage these foundations through the Curry-Howard Isomorphism [Howard, 1980] which equates proofs of propositions with programs of some type. By studying program synthesis through the lens of type theory, we hope to bring to bear the large body of work on type systems and proof search onto the program synthesis problem to enable efficient synthesis of typed, functional programs.

1.1 The Landscape of Program Synthesis

Many sub-disciplines within computer science tackle the problem of search in different ways, and as a result, these disciplines all have unique perspectives on program synthesis. The earliest such efforts came out of artificial intelligence and automatic theorem proving communities during the 60s and 70s [Green, 1969; Summers, 1976]. Since then approaches from machine learning [Briggs and O'Neill, 2008; Lau, 2001; Weimer et al., 2009], formal methods [Bodik et al., 2010; Kuncak et al., 2010; Srivastava et al., 2010], and programming language theory [Albarghouthi et al., 2013; Gvero et al., 2013; Scherer and Rèmy, 2015]

have all been applied to the program synthesis. Today, the field has seen a large resurgence in interest due to a number of factors:

1. General computational power has increased at an exponential rate over the last four decades. [Moore, 1965]

2. The rise of domain-specific languages, and more targeted programming domains such as protocols [Alur et al., 2005; Udupa et al., 2013], concurrent programs [Prountzos et al., 2012; Solar-Lezama et al., 2008; Černý et al., 2011], education [Singh et al., 2013], and strings and spreadsheets [Gulwani, 2011] has given synthesis tools smaller, more tractable domains to operate over.

3. Related to the first point, the rise of sophisticated solver technology, in particular SAT and SMT solvers [Barrett et al., 2008] have helped make once intractable problems of search more feasible in practice.

Here we briefly survey the field of program synthesis[1] to get a sense of what approaches have been previously studied and how they contrast with our own type-theoretic style. We defer discussion of how these approaches contrast with our own until after we present the details of how our type-directed program synthesis systems operate in the presence of various types (Section 2.4 and Section 5.4).

1.1.1 Methodologies

AI and Logic-based Techniques The earliest methods for program synthesis were developed from the automated theorem proving community. These researchers were motivated by the promise of generating programs from specifications that were provably correct by virtue of being derived from specification. Some methods used techniques lifted from early automated theorem prover technology, such as Green's resolution-based QA3 system, to translate programs from logical specification [Green, 1969]. These specifications took the form of complete axiomitizations of the problem space in first-order logic solved using resolution-based proof search techniques. Others took to direct rewriting tactics over the specification, for example, Manna and Waldinger's DEADALUS system which rewrote logical specification [Manna and Waldinger, 1979] and Summers's THESYS system which rewrote examples [Summers, 1976]. Later, efforts from researchers such as Manna and Waldinger sought to unify the use of provers and transformation rules [Manna and Waldinger, 1980]. Because all of these early works were rooted within theorem proving, the target language for synthesis was universally (a purely functional subset of) Lisp.

[1] Because program synthesis is such a vast field of study, we don't intend on capturing its full breadth here. For more thorough introductions to program synthesis, we recommend reading the surveys by Kreitz [1998], Flener and Yilmaz [1999], Gulwani [2010], and Kitzelmann [2010a].

These methods were technically innovative, but ultimately lacked in practicality. In particular, the logical specifications demanded by these tools were far removed from the reasoning styles that programmers understood. They were also highly constrained in the sorts of program symbols they were allowed to utilize. Finally, they were extremely heavyweight and did not scale past anything but the smallest of example programs [Kreitz, 1998].

Regardless, THESYS is particularly note worthy in that it is one of the first *inductive programming* synthesis systems [Kitzelmann, 2010a] where it is able to generate programs in the presence of *partial specification* such as examples. Examples form a partial specification because most programs of interest have an infinite range and a finite set of concrete examples can only specify a finite subset of that range. More modern systems such as IGOR2 have evolved from this line of work, and have overcome many of issues listed above [Hofmann et al., 2010; Kitzelmann, 2010b]. IGOR2 uses a combination of examples, background knowledge, and program schemes—program skeletons that capture recurring patterns of program behavior such as folds or maps—to derive target functions by discovering patterns in the examples and deriving a set of recursive rules for generating them.

In contrast, other inductive programming systems do not perform manipulation of the examples directly. Rather they employ a *guess-and-check* approach where they enumerate candidate programs and evaluate them to verify that they satisfy the examples. For example Katayama's MAGICHASKELLER [Katayama, 2012] enumerates programs according to a set of logical rules and permitted components and evaluates them against user-provided input/output examples. And Albarghouthi's ESCHER [Albarghouthi et al., 2013] builds up increasingly complicated components from atoms (*i.e.*, variables and constants) and tests whether those atoms satisfy the examples. Notably, when the system requires additional examples, such as when it evaluates a recursive function call, ESCHER queries the user to provide additional examples. LASY [Perelman et al., 2014] provides an example-driven framework for synthesizing programs in expert-written domain-specific languages. Finally, Feser's λ^2 system [Feser et al., 2015] also enumerates programs and checks them against examples. However, unlike previous efforts, they *refine* examples as they synthesize expressions, producing new examples appropriate for synthesizing the sub-expressions of this overall expression. Notably, these final approaches blur the line between the AI/logical tradition of program synthesis born from the original literature from the 70s and the more modern verification-based tradition that we see today.[2]

Note that while not directly related to program synthesis, because they explore the search space of programs through term enumeration, these guess-and-check

[2]At least, "modern" by the standards of when this thesis is written. While program synthesis has proven to be an enduring problem, the different approaches understandably come and go as computer science matures and our technology becomes more advanced.

inductive programming approaches share many concerns with *automatic test generation* [Claessen et al., 2014; Grygiel and Lescanne, 2013; Rodriguez Yakushev and Jeuring, 2010]. In particular when enumerating terms, we want to avoid generating redundant or otherwise unnecessary terms, in particular, terms that are equivalent to previously generated terms.

Machine Learning Techniques Bridging the gap between logic and machine learning are *inductive logic programming* (ILP) [Muggleton and Raedt, 1994] techniques that apply machine learning to problems expressed in first-order logic, *i.e.*, Prolog programs. Inductive logic programming is an umbrella term representing an entire sub-field of machine learning that employs this methodology for solving learning-based problems.Researchers that study inductive logic programming synthesis [Flener and Yilmaz, 1999] apply these techniques specifically to program synthesis. For example, Sankaranarayanan et al. [2008] use ILP to mine library specifications by running unit tests on a library to gather information about the operations of the library. This information is then processed using ILP to produce Prolog specifications of the library's behavior.

Other approaches rooted in machine learning have been applied to program synthesis as well. For example, genetic programming techniques have been used by Briggs and O'Neill [2008] to synthesize combinator expressions in a typed, functional programming language and by Weimer et al. [2009] to automatically locate bugs and derive patches in legacy C code. Gulwani and Jojic [2007] used probabilistic inference to synthesize imperative programs from input/output examples. And finally, Lau [2001] in her thesis developed version space algebras to synthesize text editor macros from examples—here, demonstrations by the user of the text macro they intended for the system to synthesize.

Verification-based Techniques The most recent efforts in program synthesis lie in the programming languages community. In particular, as off-the-shelf SAT and SMT solver technology like the Z3 theorem prover [De Moura and Björner, 2008] rapidly matured over the last decade, the verification community has been quick to take advantage of their power. With respect to program synthesis, this means transforming the specification given by the user into a series of constraints that can be discharged by the solver. The output of the solver can then be used to guide the search process accordingly.

The most well-known use of solver technology in program synthesis is Solar-Lezama's Sketch [Solar-Lezama, 2008]. Sketch allows users to write skeletons of Java-like programs annotated with holes whose contents are specified by generator expressions that describe the allowable set of program constructs for those holes. Sketch then translates the constraints on those holes, *e.g.*, assertions or reference implementations, into satisfiability equations which are then discharged by a SMT solver using Counterexample Guided Inductive Synthesis (CEGIS). Other work

that uses solver technology in synthesis includes Bodik et al. [2010]'s work to support incremental program develop with holes and examples, called *angelic nondeterminism* and Torlak's Rosette which supports the development of solver-aided domain specific languages [Torlak and Bodik, 2014].

In many of these situations, we can refine the problem domain sufficiently that we can restrict the syntax of allowed programs to a small subset, an approach called *syntax-guided synthesis* [Alur et al., 2013]. For example, Singh et al. [2013] use this approach in the context of generating automatic feedback for introductory programs. The restricted syntax, provided by an instructor using their system, captures the likely set of mistakes that a student might make on an assignment. Gulwani [2011] also use a restricted synthesis domain to capture string processing behavior. This synthesis technology is used, in turn, to implement the FlashFill feature of Microsoft Excel.

In addition to satisfiability solvers, other verification technology has been re-appropriated for the purposes of program synthesis. In particular, techniques that leverage types, the focus of this dissertation, have been explored to some degree. For example Djinn [Augustsson, 2004] synthesizes Haskell programs from highly refined type signatures. Prospector [Mandelin et al., 2005], Perelman's auto-completion tool for C# programs [Perelman et al., 2012], and InSynth [Gvero et al., 2013] all leverage types to accomplish code auto-completion. Most recently, λ^2 [Feser et al., 2015] uses types to refine the input/output examples that they receive, and Scherer and Rèmy [2015] phrase program synthesis in terms of type inhabitation.

1.2 Program Synthesis With Types

From Section 1.1, we see that a multitude of approaches to program synthesis have been previously explored, each with their own strengths and weaknesses. However, no single system has the combination of features necessary to fully capture the type-directed programming style that motivated our journey into program synthesis to begin with. In particular, this style requires support for higher-order functions, recursive functions, and algebraic data types. Many of the prior systems focus on languages that do not support one of more of these features as they are based on variants of Lisp, C, or Java. Furthermore, many of the systems that use solver technology are not capable of handling higher-order and/or recursive data as these solvers work over first-order logics. Of the systems that most closely target this space of language features:

- Escher [Albarghouthi et al., 2013] synthesizes recursive functions using input/output examples, but in a Lisp-like untyped setting.
- Leon [Kuncak et al., 2010] and Igor2 [Hofmann et al., 2010] synthesize recursive functions using input/output examples over algebraic data types,

but they cannot handle higher-order functions. In particular, LEON's reliance on solver technology keeps them from handling higher-order functions. IGOR2 allows for usage of higher-order function components but it does not appear to synthesize functions that take higher-order functions as arguments.

- λ^2 [Feser et al., 2015] synthesizes recursive functions over algebraic data types using input/output examples. However, is is entirely component driven—λ^2 only synthesizes the composition of function applications efficiently—and cannot pattern match over algebraic data types.

Furthermore, it is not clear whether any of these systems scale up to richer type systems such as linear types [Girard, 1987], refinement types [Freeman and Pfenning, 1991], and dependent types [Martin-Löf, 1984].

1.2.1 Foundations

To address these concerns, we create a theoretical foundation for program synthesis using types. With this foundation, we answer several key questions:

- How can we take advantage of rich types in order to prune the search space of possible programs?
- How can we use types to incremental refine the specification provided by the user in step with the program that we synthesize?
- What are the meta-theoretic properties of synthesis systems based on types? In particular, are they sound and complete?

The basis of our foundation is a technique for transforming a programming language's type system into a type-directed, example-powered program synthesis system for that language. This transformation allows us to gain immediate insight into how to synthesize programs and refine examples of particular types. In some cases, this insight alone is sufficient to integrate a new type into our synthesis systems; in others, we must address additional issues that arise when synthesizing programs of these types.

We begin by applying this technique to the simplest type system possible, the simply-typed lambda calculus (Chapter 2). The resulting *synthesis calculus*, $\lambda^{\rightarrow}_{syn}$, allows us to explore in detail the transformation process as well as how these resulting synthesis systems operate. We then begin integrating additional types into the mix with the goal of arriving at a synthesis system for a more realistic typed, functional programming language. First, we consider simple extensions to the simply-typed lambda calculus—products, records, and sums (Chapter 4). And then, we build more complex synthesis calculi to handle complex type extensions—ML_{syn} to handle recursion with algebraic data types (Chapter 5) and $\lambda^{\forall}_{syn}$ to handle polymorphism (Chapter 9).

1.2.2 Metatheory

We use our core calculi—$\lambda^{\rightarrow}_{syn}$, ML$_{syn}$, and $\lambda^{\forall}_{syn}$—to study the meta-theoretic properties of program synthesis with types. In particular, we are concerned with two key properties of synthesis systems:

- **Soundness**: does the synthesized program obey our input specification?
- **Completeness**: can we synthesize all programs?

We first study the soundness and completeness of $\lambda^{\rightarrow}_{syn}$ in full detail (Chapter 3). By doing so, we extract the key lemmas that we must prove to show that soundness and completeness holds of the whole synthesis calculus. We then prove these lemmas for our simple extensions to $\lambda^{\rightarrow}_{syn}$ (Chapter 4) as well as polymorphism (Chapter 9). ML$_{syn}$ proves to be much more complex due to recursion, so we also investigate its metatheory in full detail (Chapter 6).

1.2.3 Implementation

The synthesis calculi we develop in our work give us a basic understanding of the design and behavior of type-directed synthesis systems, but they are not practical synthesis algorithms as-is because they are both unoptimized and highly nondeterministic. Consequently, in addition to understanding type-directed program synthesis from a theoretical perspective, we would like to know empirically how these systems behave on real-world examples.

To do this, we take the calculus closest to a real-world typed, functional programming language, ML$_{syn}$, and transform it into an actual program synthesizer, Myth, for a core subset of the OCaml programming language (Chapter 7). Because our program synthesizer is type-theoretic, we are able to adapt several proof search techniques for our domain—various caching schemes and search pruning heuristics—to greatly optimize the synthesis procedure. Furthermore, thanks to our theoretical foundations, we are able to analyze the impact of these optimizations on the soundness and completeness of our system. Finally, we evaluate Myth's effectiveness on a benchmark suite of functional programs and explore Myth's behavior on these examples (Chapter 8).

1.2.4 Statement of Contribution

The synthesis calculi for ML-like programs, ML$_{syn}$, that we discuss in Chapter 5 as well as the implementation, Myth, that we develop in Chapter 7 and evaluate in Chapter 8 were originally presented in PLDI 2015 [Osera and Zdancewic, 2015]. The chapters listed above constitute a greatly expanded presentation of these two artifacts. The presentation of tuples and records in Chapter 4 in our simply-typed synthesis calculus, $\lambda^{\rightarrow}_{syn}$, was adapted from Frankle [2015] and Shah [2015],

respectively, who originally integrated these features into ML_{syn}. The remaining content is original work.

Chapter 2

A Simple Synthesis Calculus

Program synthesis is an undecidable problem, so when building practical synthesis tools, we must resort to approximations and heuristics to not only make the problem solvable, but also tractable. However, this reality is frequently at odds with building a system that is understandable, as the approximations and heuristics frequently bleed together so that their individual contributions are not clear. Ideally, we would like to build synthesis systems in such a way that we know precisely the impact of each feature and design choice. For example, does ruling out a certain class of programs impact the completeness of the synthesis algorithm? If so, are these programs relevant or are they safe to ignore for practical purposes?

To this end, we begin by constructing a theoretical foundation for program synthesis with types. In typical programming language theory style, we strip away everything but the essential components of typed, functional programming languages by starting with the simply-typed lambda calculus, $\lambda^{\to}$. From $\lambda^{\to}$, we build a generator for well-typed λ-terms and then integrate a notion of specification into the system to create a program synthesis calculus, $\lambda^{\to}_{syn}$.

$\lambda^{\to}_{syn}$ itself is far removed from a usable implementation of a program synthesizer. However, it allows us to deeply study how to synthesize programs with types, and the procedure's meta-theoretic properties. For example, the synthesis judgment of $\lambda^{\to}_{syn}$ is sound and complete with respect to synthesis. In successive chapters, we evolve $\lambda^{\to}_{syn}$ into a practical program synthesizer for ML-like programs, sacrificing some of these properties to handle advanced types and gain efficiency.

2.1 Generating λ-terms

Figure 2.1 gives the syntax and semantics for $\lambda^{\to}$ which contains the essence of a typed, functional programming language—variables, functions, and application—and their usual type checking rules. In addition to these essentials, we also augment $\lambda^{\to}$ with a base type T and a finite, non-empty set of constants $c_1, \ldots, c_k$ of type T. While multiple constants are not necessary—a single constant makes

$$
\begin{array}{lcll}
\tau & ::= & \tau_1 \to \tau_2 \mid T & \text{Types} \\
e & ::= & x \mid e_1\, e_2 \mid \lambda x{:}\tau.\, e \mid c & \text{Terms} \\
v & ::= & \lambda x{:}\tau.\, e \mid c & \text{Values} \\
\mathcal{E} & ::= & \square \mid \mathcal{E}\, e \mid v\, \mathcal{E} & \text{Evaluation Contexts} \\
\Gamma & ::= & \cdot \mid x{:}\tau, \Gamma & \text{Typing Contexts}
\end{array}
$$

$$
\boxed{\Gamma \vdash e : \tau}
\qquad
\textsc{t-var}\ \frac{x{:}\tau \in \Gamma}{\Gamma \vdash x : \tau}
\qquad
\textsc{t-app}\ \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\, e_2 : \tau_2}
$$

$$
\textsc{t-lam}\ \frac{x{:}\tau_1, \Gamma \vdash e : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.\, e : \tau_1 \to \tau_2}
\qquad
\textsc{t-base}\ \frac{}{\Gamma \vdash c : T}
$$

$$
\boxed{e \longrightarrow e'}
\qquad
\textsc{eval-ctx}\ \frac{e \longrightarrow e'}{\mathcal{E}[e] \longrightarrow \mathcal{E}[e']}
\qquad
\textsc{eval-app}\ (\lambda x{:}\tau.\, e)\ v \longrightarrow [v/x]e
$$

Figure 2.1: $\lambda^{\to}$ definition

T isomorphic to the standard Unit type—they allow us to synthesize interesting programs. We also define a small-step, call-by-value operational semantics for the language using evaluation contexts $\mathcal{E}$ which capture the standard congruence rules for call-by-value evaluation.

To generate $\lambda^{\to}$ terms, we could simply enumerate terms according to the grammar given in Figure 2.1, perhaps in order of increasing term size. We'll define the size of a term to be the number of abstract syntax tree (AST) nodes used to represent the term. For example, the term $\lambda x{:}T.\, y\, x$ has size 5 because the type T has size one, the application $y\ x$ has size 3, and the lambda itself contributes one additional AST node.

However, simple enumeration is not very practical, especially when we have the type system at our disposal. To see why, consider the number of closed $\lambda^{\to}$-terms—terms generated from the empty typing context—at a given size. Figure 2.2 gives the number of such untyped terms, well-typed terms, and well-typed terms at type T when there are two constants c_1 and c_2 of type T. For example:

- $c_1\ (\lambda x{:}T.\, x)$ is a syntactically valid, yet ill-typed term of size five.
- c_2 is a well-typed term of type T and size one.
- $(\lambda x{:}T \to T.\, x)\ (\lambda x{:}T.\, x)$ is a well-typed term of type $T \to T$ of size nine.

Noting that the y-axis scale of Figure 2.2 is at logscale, we can see that the number of syntactically valid, but not necessarily well-typed terms is staggering. Even in this extremely limited language, there are 11,084,176 such terms at size 15!

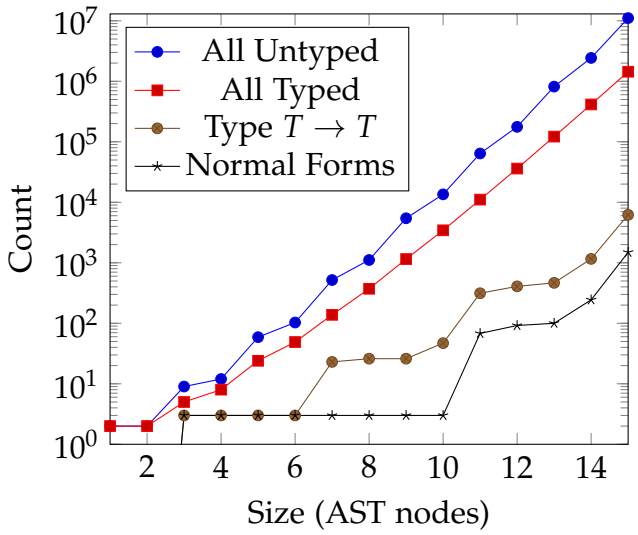


Figure 2.2: Number of closed untyped terms, typed terms, terms of type $T \to T$, and normal forms of type $T \to T$ in $\lambda^{\to}$ with two constants of type T.

In contrast, if we limit ourselves to only the well-typed terms, we save an order of magnitude of work; there are only 1,439,481 well-typed terms of size 15 in $\lambda^{\to}$. Finally, if we consider only well-typed programs, we may as well further refine our search to programs of a particular type that we are interested in. The savings here is even larger; if we are trying to find terms of type $T \to T$, we only have to search 6,205 terms.

We could refine our term generation strategy by enumerating syntactically well-typed terms and then filtering out terms that fail to type check. This has the benefit of ensuring that any terms that we keep around will be well-typed. However, we still pay the time and space overhead of generating such terms and then type checking them, even if we end up throwing them away in the end. This is undesirable in the presence of the steep combinatorial explosion of terms as their size increases, for example, Grygiel and Lescanne [2013] show that there are approximately 2^{219} closed-terms in the simply-typed lambda calculus of size 50. Rather than type check after the fact, we should *integrate* type checking into term enumeration so that we only ever consider well typed terms.

To arrive at such an algorithm, we start with the type checking relation presented in Figure 2.1:

$$\Gamma \vdash e : \tau.$$

When implementing a type checker based on this relation, we note that Γ and e serve as inputs and τ serves as either an input or output. To derive a type-aware term enumeration system from this judgment, let's simply flip the inputs and outputs: Γ and τ will be inputs and the output will be an e. The resulting relation,

$$\boxed{\Gamma \vdash \tau \rightsquigarrow e}$$

$$\textsc{gen-var}\ \frac{x{:}\tau \in \Gamma}{\Gamma \vdash \tau \rightsquigarrow x} \qquad \textsc{gen-app}\ \frac{\Gamma \vdash \tau_1 \to \tau_2 \rightsquigarrow e_1 \qquad \Gamma \vdash \tau_1 \rightsquigarrow e_2}{\Gamma \vdash \tau_2 \rightsquigarrow e_1\, e_2}$$

$$\textsc{gen-lam}\ \frac{x{:}\tau_1, \Gamma \vdash \tau_2 \rightsquigarrow e}{\Gamma \vdash \tau_1 \to \tau_2 \rightsquigarrow \lambda x{:}\tau_1.\, e} \qquad \textsc{gen-base}\ \frac{}{\Gamma \vdash T \rightsquigarrow c}$$

Figure 2.3: $\lambda^{\to}$ typed term generation

$\Gamma \vdash \tau \rightsquigarrow e$, describes the possible ways we can generate a term e of type τ.

Figure 2.3 gives the rules for $\lambda^{\to}$ well-typed term enumeration. A complete derivation in this judgment corresponds to how we generate a particular term of a given type. Therefore, searching for all the valid derivations for a particular Γ and τ is equivalent to enumerating all programs of type τ under Γ.

As a concrete example, we generate the well-typed closed term $(\lambda x{:}T \to T.\, x)\ (\lambda x{:}T.\, c_1)$ through the following derivation:

$$\frac{\dfrac{\dfrac{x{:}T \to T \in x{:}T \to T}{x{:}T \to T \vdash T \to T \rightsquigarrow x}}{\cdot \vdash (T \to T) \to (T \to T) \rightsquigarrow (\lambda x{:}T \to T.\, x)} \qquad \dfrac{\dfrac{}{x{:}T \vdash T \rightsquigarrow c_1}}{\cdot \vdash (T \to T) \rightsquigarrow (\lambda x{:}T.\, c_1)}}{\cdot \vdash T \to T \rightsquigarrow (\lambda x{:}T \to T.\, x)\ (\lambda x{:}T.\, c_1)}$$

Note that the derivation for term enumeration is identical to the derivation for type checking! This is because the rules for well-typed term enumeration mirror the rules for type checking.[1] This observation leads to two natural properties of our term enumeration judgment:

Theorem 2.1.1 (Soundness of $\lambda^{\to}$ term enumeration). *If $\Gamma \vdash \tau \rightsquigarrow e$ then $\Gamma \vdash e : \tau$.*

Theorem 2.1.2 (Completeness of $\lambda^{\to}$ term enumeration). *If $\Gamma \vdash e : \tau$ then $\Gamma \vdash \tau \rightsquigarrow e$.*

Proof. Both theorems follow from straightforward inductions on the given term-enumeration or type-checking derivations, respectively. □

---

[1] In fact, because the type-checking and term-enumeration judgments describe *relations* which do not have inputs or outputs, they are equivalent! But then, why did we bother including a term-enumeration judgment at all? We use this term-enumeration judgment as a starting point to arrive at a synthesis judgment that will do more than relate terms and types; it will relate examples as well.

Theorem 2.1.1 states that enumerated terms are well-typed, and Theorem 2.1.2 states that we are able to enumerate the well-typed terms. By combining both theorems, we know that well-typed term enumeration produces *exactly* the set of well-typed terms, *i.e.*, the "All Typed" dataset from Figure 2.2.

2.1.1 Enumerating Normal Forms

By restricting term generation to well-typed terms, we substantially reduce the search space of programs. However there are still many redundant terms that we could generate. For example, the following terms are redundant:

- $(\lambda x{:}T.\, x)\, c_1$
- $(\lambda x{:}T \to T.\, x)\, (\lambda y{:}T.\, y)$.

These terms evaluate, according to our call-by-value semantics, to the values c_1 and $\lambda y{:}T.\, y$, respectively. In general, any term that is not a value, *i.e.*, not in *normal form*, is redundant with the value that it evaluates to. We say that the two terms are *β-equivalent*, taking the standard definition of β-equivalence as the smallest equivalence relation that contains our evaluation relation ($\longrightarrow$).[2]

By restricting our search to these normal forms, we narrow the search space of programs even further! Returning to Figure 2.2, we see that the number of closed normal forms of size 15 and type $T \to T$ is another order of magnitude smaller than the number of terms of type $T \to T$ (1,489 versus 6,205 such terms). To do this, we restrict terms so that they cannot contain pending β-reductions. In $\lambda^{\to}$, the only β-reduction available is function application which applies whenever we have a term of the form $(\lambda x{:}\tau.\, e)\, v$. We avoid this situation by splitting expressions into two sets of sub-terms.

$$\begin{aligned} E &::= x \mid E\, I \\ I &::= E \mid \lambda x{:}\tau.\, I \mid c \end{aligned}$$

E-forms include variables and *elimination* forms, and I-forms include *introduction* forms. For example, lambdas introduce values of arrow type, and function application eliminates these values. In contrast, the base type T is introduced by its constants $c_1, \ldots, c_k$ and have no corresponding elimination forms.

With this syntax, I and E terms are in normal form by construction! To see this, note that function application requires that the head of a function application is an E, and the syntax of Es bottoms out at variables. Therefore, all applications will be of the form $x\, I_1 \ldots I_m$ where a variable is always at the head of the application. Arguments, in contrast, are allowed to be any I-term which includes Es because Es are included in the definition of I.

[2]This is true of $\lambda^{\to}$ because it is strongly normalizing. In the presence of non-termination, this does not hold because infinite loops do not have a corresponding normal form.

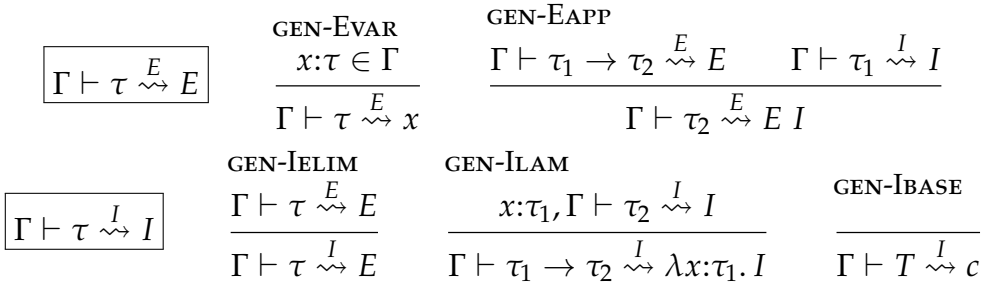


Figure 2.4: $\lambda^{\to}$ typed normal-form term generation

This technique of splitting the syntax of expressions into introduction and elimination forms has several useful effects. In proof search, this syntax is useful for generating normal form proofs which, by the Curry-Howard Isomorphism [Howard, 1980], is equivalent to generating normal form programs in our setting. Dividing the syntax in this manner also facilities a description of *bidirectional type checking* [Pierce and Turner, 2000] where we explicitly state when a type is an input or output in the type checking judgment. This distinction becomes very useful when we talk about refining specification in the context of full program synthesis in Section 2.3.

Figure 2.4 gives the rules for generating $\lambda^{\to}$ terms in this restricted term language. Because there are two syntactic forms, we introduce two judgments to synthesize E- and I-terms, $\overset{E}{\rightsquigarrow}$ and $\overset{I}{\rightsquigarrow}$, respectively. Note that the judgments are mutually recursive because the definitions of E and I terms are mutually recursive. The rule GEN-IELIM allows us to generate an E where ever an I is expected (because Es are a proper subset of the Is). And the rule GEN-EAPP generates the function as an E and its argument as an I. Otherwise, the rules are identical to the e-term generation judgment given in Figure 2.3.

2.2 Specifying λ-terms

With well-typed term enumeration, we can efficiently search the space of $\lambda^{\to}$ programs. In addition to this search method, we also require some way of *specifying* which particular programs we would like to find during the synthesis process. Specification can take many forms, for example, logical properties [Kuncak et al., 2010; Solar-Lezama, 2008] or execution traces [Lau, 2001]. Here, we will consider specifying programs with a combination of *types* and *examples*, in particular, input/output pairs of the form $(v \Rightarrow v')$ that say a synthesized function should produce the value v' when given the value v.

2.2.1 Examples as Specification

There are trade-offs involved in choosing examples as our means of specification over other options. Examples typically act as incomplete specifications—for example, a finite set of input/output example pairs can only specify a finite subset of a function's behavior. This is a significant limitation because many non-equivalent programs satisfy a set of examples, and the synthesizer must choose among them by inferring the intent of the user. For example, if the synthesizer is provided with the single input/output example

$$c_1 \Rightarrow c_2 \Rightarrow c_1$$

that states the synthesized program should return c_1 when given c_1 and c_2 as input, it isn't clear if the user wanted the program that always selects its first argument

$$\lambda x{:}T.\, \lambda y{:}T.\, x,$$

or the program that always returns c_1,

$$\lambda x{:}T.\, \lambda y{:}T.\, c_1.$$

or some more elaborate function. To solve this problem, the user must specify more examples which is burdensome, or the synthesizer must resort to heuristics to choose a final program, *e.g.*, choosing the smallest program with the fewest number of variables. Thankfully, heuristics like this tend to work well in practice. Furthermore, this ambiguity can be mitigated by smart interface design decisions, *e.g.*, displaying the top five synthesis results rather than a singular result and allowing the user to choose.

In exchange for being incomplete, examples can be less burdensome for the user to specify. Because examples are typically concrete instances of a function's behavior, they allow the user to reason more about *what* their function ought to do rather than *how* the function ought to go about doing it. This is favorable because the "how" is usually the sticking point for a user to create a program on their own, whether it is because they do not understand the syntax of programming language or the algorithms necessary to produce the desired behavior.

For example, consider the set of input/output examples specifying a function in a language extended with natural numbers and lists:

$$\begin{aligned} [] &\Rightarrow [] \\ [0] &\Rightarrow [0] \\ [0,1] &\Rightarrow [1,0] \\ [0,1,2] &\Rightarrow [2,1,0] \end{aligned}$$

From this limited set of examples, the user's intent seems obvious: they want to

synthesize a function that sorts a list of natural numbers. In contrast, consider a logical specification of how this function ought to behave:

$$\mathsf{sorted}(l) = \forall x :: l' \sqsubseteq l.\ \forall y \in l.\ x < y$$

The sorted predicate states that for all sub-lists ($\sqsubseteq$) of l, the head x of that sub-list is the least element of that sub-list. While this logical specification more precisely describes the behavior of any function that sorts a list, it takes a bit of ingenuity to come up with this specification.

Finally and most importantly, we use examples because they decompose naturally with types. For example, the input/output example $[0,1] \Rightarrow [1,0]$ decomposes according to the arrow type list $\rightarrow$ list. We know that when the synthesized function's argument has the value $[0,1]$, the body of that function must evaluate to $[1,0]$. In contrast, it is not clear how to decompose the sorted predicate according to the type of the function we are synthesizing.

2.2.2 Example Specification in $\lambda^{\rightarrow}$

To adopt example specifications into $\lambda^{\rightarrow}$, we need to account for the presence of higher-order functions. Therefore, we adopt the following grammar of *example values* χ that we use as our specification:

$$\chi ::= c \mid \overline{v_i \Rightarrow \chi_i}^{i<m}$$

For each type of $\lambda^{\rightarrow}$, we include a term that stands in as an example of that type. As we extend our language with additional features, we'll find that most of the time this term is simply the value form for that type, *e.g.*, the constants c_k for base type T. However, lambdas serve as a poor example value for arrow types because providing a function value is tantamount to describing exactly how the function ought to behave! Instead, we make sets of input/output pairs first-class example values with *partial function* terms written:

$$v_1 \Rightarrow \chi_1 \mid \ldots \mid v_m \Rightarrow \chi_m$$

A partial function introduces m input/output pairs of the form $v_i \Rightarrow \chi_i$ that specify that when the synthesized function is supplied a v_i, it produces a χ_i. Like $\rightarrow$, $\Rightarrow$ associates to the right, and as a result, functions of k arguments are represented in curried style, $v_1 \Rightarrow \ldots \Rightarrow v_k \Rightarrow \chi$. Because of this, we restrict the right-hand side of a $\Rightarrow$ to be an example value. This ensures that lambdas never appear as the *goal* of a synthesis problem. In contrast, the left-hand side of a $\Rightarrow$ is a regular value which includes lambdas (and not partial functions). This means that when we give examples for higher-order functions, we provide function values for the arguments of those functions. We disallow partial functions as examples for

higher-order arguments as doing so has surprising effects on the metatheory of the system; we explore this in more detail in Chapter 6 where we lift this restriction to handle more advanced language features. For shorthand, we write the above partial function as $\overline{v_i \Rightarrow \chi_i}^{i<m}$ where the partial function contains m input/output pairs.

As a concrete example within $\lambda^{\rightarrow}$, consider the following synonyms for the church encodings of booleans over base type T:

$$\begin{aligned} \mathsf{bool} &\stackrel{\text{def}}{=} T \rightarrow T \rightarrow T \\ \mathsf{true} &\stackrel{\text{def}}{=} \lambda t{:}T.\, \lambda f{:}T.\, t \\ \mathsf{false} &\stackrel{\text{def}}{=} \lambda t{:}T.\, \lambda f{:}T.\, f \end{aligned}$$

Then the following partial function:

$$\mathsf{true} \Rightarrow c_1 \Rightarrow c_2 \Rightarrow c_1 \;\mid\; \mathsf{false} \Rightarrow c_1 \Rightarrow c_2 \Rightarrow c_2$$

specifies the if function of type $\mathsf{bool} \rightarrow T \rightarrow T \rightarrow T$, usually defined as:

$$\mathsf{if} \stackrel{\text{def}}{=} \lambda b{:}\mathsf{bool}.\, \lambda t{:}T.\, \lambda f{:}T.\, b\, t\, f$$

Note that the convenience of specifying arguments to higher-order functions as lambdas allows us to use true and false directly in the input/output examples.

2.3 λ_{syn}: A Program Synthesis Calculus

So far, we modified the type checking judgment of $\lambda^{\rightarrow}$ to create a type-directed term enumeration judgment and defined a grammar of example values to serve as our specification. Let's combine these two into a *synthesis calculus*, $\lambda^{\rightarrow}_{syn}$, that synthesizes $\lambda^{\rightarrow}$ terms.

2.3.1 Integrating Search and Specification

How do we use our example values in tandem with term enumeration? A simple strategy is to simply enumerate terms as per Figure 2.4 and check each term to see if it satisfies the example values. However, we waste effort with this approach because example values ought to help rule out programs *during* term generation rather than help us filter programs after the fact. For example, if we have following input/output pairs in a partial function:

$$\ldots \;\mid\; c_1 \Rightarrow c_2 \;\mid\; c_2 \Rightarrow c_1 \;\mid\; \ldots$$

We should never need to generate the constant functions $\lambda x{:}T.\, c_1$ and $\lambda x{:}T.\, c_2$ because the examples rule them out.

To develop a more robust strategy that uses examples in a productive way, let's imagine how we might synthesize a function by hand using input/output examples. Consider synthesizing the if function of type $\mathsf{bool} \to T \to T \to T$ that we discussed in Section 2.2.2 where bool is shorthand for the type $T \to T \to T$. We were given the partial function

$$\mathsf{true} \to c_1 \to c_2 \to c_1 \;\mid\; \mathsf{false} \to c_1 \to c_2 \to c_2$$

as our example value with true and false of type bool serving as shorthand for $\lambda t{:}T.\, \lambda f{:}T.\, t$ and $\lambda t{:}T.\, \lambda f{:}T.\, f$, respectively.

Assuming that we have no top-level bindings to work with (*i.e.*, we are operating in a closed context), we know from the desired goal type, $\mathsf{bool} \to T \to T \to T$, that the synthesized program should have the shape:

$$\lambda b{:}\mathsf{bool}.\, \lambda t{:}T.\, \lambda f{:}T.\, \blacksquare$$

where the body of the function, denoted by the hole $\blacksquare$, must be filled in with an expression of type T. When considering what to fill in for $\blacksquare$, we note that our two input/output examples from our partial function give rise to two possible states:

$$b = \mathsf{true}, t = c_1, f = c_2, \blacksquare = c_1$$
$$b = \mathsf{false}, t = c_1, f = c_2, \blacksquare = c_2$$

Each state or *example world* corresponds to an imaginary execution of the function being synthesized using each of the input/output examples. The first world corresponds to the example where we select the True argument, and the second world corresponds to the example where we select the False argument. In each example world, we keep track of the value each argument is bound to as well as what the function should produce: $\blacksquare = c_1$ and $\blacksquare = c_2$, respectively.

With all this information, the correct choice of program to fill in the hole is clear. The term $b\, t\, f$ has the correct type and satisfies our examples because we can substitute the values bound to the variables in each example world and evaluate to obtain

$$\mathsf{true}\, c_1\, c_2 \longrightarrow^* c_1$$
$$\mathsf{false}\, c_1\, c_2 \longrightarrow^* c_2$$

which are the values that each example world required.

$$
\begin{array}{lcll}
\tau & ::= & \tau_1 \to \tau_2 \mid T & \text{Types} \\
e & ::= & x \mid e_1\, e_2 \mid \lambda x{:}\tau.\, e \mid c & \text{Terms} \\
v & ::= & \lambda x{:}\tau.\, e \mid c & \text{Values} \\
\mathcal{E} & ::= & \square \mid \mathcal{E}\, e \mid v\, \mathcal{E} & \text{Evaluation Contexts} \\
\Gamma & ::= & \cdot \mid x{:}\tau, \Gamma & \text{Typing Contexts} \\
 & & & \\
E & ::= & x \mid E\, I & \text{Elimination Terms} \\
I & ::= & E \mid \lambda x{:}\tau.\, I \mid c & \text{Introduction Terms} \\
\sigma & ::= & \cdot \mid [v/x]\sigma & \text{Evaluation Environments} \\
\chi & ::= & c \mid \rho & \text{Example Values} \\
\rho & ::= & \overline{v_i \Rightarrow \chi_i}^{i<m} & \text{Partial Functions} \\
X & ::= & \cdot \mid \sigma \mapsto \chi, X & \text{Example Worlds}
\end{array}
$$

$\boxed{\Gamma \vdash e : \tau}$ $\boxed{e \longrightarrow e'}$ (Same as $\lambda^{\to}$)

$\boxed{\Gamma \vdash E \Rightarrow \tau}$

$$\textsc{t-Evar}\ \frac{x : \tau \in \Gamma}{\Gamma \vdash x \Rightarrow \tau} \qquad \textsc{t-Eapp}\ \frac{\Gamma \vdash E \Rightarrow \tau_1 \to \tau_2 \qquad \Gamma \vdash I \Leftarrow \tau_1}{\Gamma \vdash E\ I \Rightarrow \tau_2}$$

$\boxed{\Gamma \vdash I \Leftarrow \tau}$

$$\textsc{t-Ilam}\ \frac{x{:}\tau_1, \Gamma \vdash I \Leftarrow \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.\, I \Leftarrow \tau_1 \to \tau_2} \qquad \textsc{t-Ibase}\ \frac{}{\Gamma \vdash c \Leftarrow T} \qquad \textsc{t-Ielim}\ \frac{\Gamma \vdash E \Rightarrow \tau}{\Gamma \vdash E \Leftarrow \tau}$$

$\boxed{\Gamma \vdash \chi : \tau}$

$$\textsc{t-ex-base}\ \frac{}{\Gamma \vdash c : T} \qquad \textsc{t-ex-pf}\ \frac{\overline{\Gamma \vdash v_i : \tau_1}^{i<m} \qquad \overline{\Gamma \vdash \chi_i : \tau_2}^{i<m}}{\Gamma \vdash \overline{v_i \Rightarrow \chi_i}^{i<m} : \tau_1 \to \tau_2}$$

$\boxed{\Gamma \vdash \sigma}$

$$\textsc{t-env-empty}\ \frac{}{\Gamma \vdash \cdot} \qquad \textsc{t-env-cons}\ \frac{x{:}\tau \in \Gamma \qquad \vdash v : \tau \qquad \Gamma \vdash \sigma}{\Gamma \vdash [v/x]\sigma}$$

$\boxed{\Gamma \vdash X : \tau}$

$$\textsc{t-exw-empty}\ \frac{}{\Gamma \vdash \cdot : \tau} \qquad \textsc{t-exw-cons}\ \frac{\Gamma \vdash \sigma \qquad \forall x{:}\tau \in \Gamma.\, \exists v.\, [x/v] \in \sigma \qquad \Gamma \vdash \chi : \tau \qquad \Gamma \vdash X : \tau}{\Gamma \vdash \sigma \mapsto \chi, X : \tau}$$

Figure 2.5: $\lambda^{\to}_{syn}$ syntax and typechecking

2.3.2 Introducing λ_{syn}

In the previous example, we performed a mixture of *symbolic evaluation* and *type-directed refinement* to synthesize the program. We now formalize this process within $\lambda^{\rightarrow}_{syn}$, a core calculus for program synthesis with types.

Figure 2.5 gives the syntax of $\lambda^{\rightarrow}_{syn}$ which features an external language of standard expressions e taken from $\lambda^{\rightarrow}$ and an internal language that splits up expressions into introduction forms I and elimination forms E. The internal language is a subset of the external language. As described in Section 2.1.1, it is precisely the normal forms of the external language that we synthesize.

Type checking and evaluation in the external language is the same as $\lambda^{\rightarrow}$. We provide additional rules for type checking the additional constructs introduced in $\lambda^{\rightarrow}_{syn}$. In particular, E and I terms are checked in a bidirectional type checking style [Pierce and Turner, 2000]. We note that during regular type checking, E terms *generate* types (as an output), and I terms *check* against types (as an input). To make this explicit, we separate E and I type checking into two separate judgments where we generate types $(E \Rightarrow \tau)$ in the former case and check against types in $(I \Leftarrow \tau)$ in the latter case. For example when type checking a variable (T-EVAR), we can extract the type of that variable from the context. In contrast, when type checking a function (T-ILAM), while we know the type of the argument from the ascription, we have no information about the body, so we must check the body against a given input type. Type checking examples values χ, in contrast, is straightforward. Constants c_k all have type T (T-EX-BASE), and a partial function $\overline{v_i \Rightarrow \chi_i}^{i<m}$ has type $\tau_1 \rightarrow \tau_2$ if all of the v_i have type τ_1 and all of the χ_i have type τ_2 (T-EX-PF).

2.3.3 Example Worlds

Example worlds were a critical component of the synthesis process for the if function in Section 2.3.1. Recall that each example world contains not only an example value that the synthesized program must evaluate to, but value bindings for each of the free variable of the program. To codify this, we define an example world as a pair $\sigma \mapsto \chi$ of a *goal example value* χ and an *environment* σ that maps variables to values. A collection of these example worlds is denoted with X, and when it is unambiguous to do so, we refer to these collections of example worlds as our "examples".

When lifting typechecking of example values to example worlds, written $\Gamma \vdash X : \tau$, we ensure (via T-EXW-CONS) that for each pair $\sigma \mapsto \chi$ that χ has type τ but also that σ is well-typed. For an environment to be well-typed, it suffices that for each binding in σ to be well-typed according to the type recorded in Γ (T-ENV-CONS). In addition, T-EXW-CONS requires that each σ contains a binding for each variable bound in Γ. This, coupled with T-ENV-CONS, gives us the following consistency principle for example worlds:

$$\boxed{\Gamma \vdash \tau \rightsquigarrow E} \qquad \textsc{eguess-var}\ \frac{x{:}\tau \in \Gamma}{\Gamma \vdash \tau \rightsquigarrow x} \qquad \textsc{eguess-app}\ \frac{\Gamma \vdash \tau_1 \to \tau_2 \rightsquigarrow E \qquad \Gamma \vdash \tau_1 \rhd \cdot \rightsquigarrow I}{\Gamma \vdash \tau_2 \rightsquigarrow E\ I}$$

$$\boxed{\Gamma \vdash \tau \rhd X \rightsquigarrow I} \qquad \textsc{irefine-arr}\ \frac{\begin{array}{c} X = \sigma_1 \mapsto \rho_1, \ldots, \sigma_n \mapsto \rho_n \\ X' = \mathbf{apply}(x, \sigma_1, \rho_1) +\!\!+ \ldots +\!\!+ \mathbf{apply}(x, \sigma_n, \rho_n) \\ x{:}\tau_1, \Gamma \vdash \tau_2 \rhd X' \rightsquigarrow I \end{array}}{\Gamma \vdash \tau_1 \to \tau_2 \rhd X \rightsquigarrow \lambda x{:}\tau_1.\, I}$$

$$\textsc{irefine-base}\ \frac{X = \sigma_1 \mapsto c_k, \ldots, \sigma_n \mapsto c_k}{\Gamma \vdash T \rhd X \rightsquigarrow c_k} \qquad \textsc{irefine-guess}\ \frac{\Gamma \vdash \tau \rightsquigarrow E \qquad E \vDash X}{\Gamma \vdash \tau \rhd X \rightsquigarrow E}$$

$$\boxed{I \vDash X} \qquad \textsc{satisfies}\ \frac{\forall \sigma \mapsto \chi \in X.\, \sigma(I) \longrightarrow^* v \wedge v \simeq \chi}{I \vDash X}$$

$$\boxed{v \simeq \chi} \qquad \textsc{eq-ctor}\ \frac{}{c_k \simeq c_k} \qquad \textsc{eq-lam-pf}\ \frac{\forall i \in 1, \ldots, m.\, (\lambda x{:}\tau.\, e)\ v_i \longrightarrow^* v \wedge v \simeq \chi_i}{\lambda x{:}\tau.\, e \simeq \overline{v_i \Rightarrow \chi_i}^{i<m}}$$

$$\mathbf{apply}(x, \sigma, \overline{v_i \Rightarrow \chi_i}^{i<m}) = [v_1/x]\sigma \mapsto \chi_1, \ldots, [v_m/x]\sigma \mapsto \chi_m$$

Figure 2.6: $\lambda^{\to}_{syn}$ synthesis and equivalence

Lemma 2.3.1 (Consistency of Example Worlds). *If* $\Gamma \vdash X : \tau_1$, *then* $\forall \sigma \mapsto \chi \in X.\, x{:}\tau \in \Gamma \leftrightarrow ([x/v] \in \sigma \wedge \Gamma \vdash v : \tau)$.

Proof. T-EXW-CONS enforces the $\to$ direction, and T-ENV-CONS enforces the $\leftarrow$ direction. □

Because our synthesis strategy requires evaluation, Lemma 2.3.1 is necessary to ensure that every free variable has a (well-typed) value.

2.3.4 Synthesis in $\lambda^{\to}_{syn}$

In $\lambda^{\to}_{syn}$, we further refine the term enumeration judgment we derived in Section 2.1 to create a full-fledged synthesis judgment that takes advantage of example values as specification. We break up synthesis into two judgments as shown in Figure 2.6:

- $\Gamma \vdash \tau \rightsquigarrow E$ (EGUESS): guess an E of type τ.
- $\Gamma \vdash \tau \rhd X \rightsquigarrow I$ (IREFINE): refine and synthesize an I of type τ that agrees with examples X.

The synthesis problem in $\lambda^{\rightarrow}_{syn}$ can be characterized as follows:

> Given a context Γ, goal type τ, and examples X where $\Gamma \vdash X : \tau$, synthesize a program I of type τ that satisfies the examples X.

The two mutually recursive judgments combine to form a non-deterministic synthesis system where a complete IREFINE derivation corresponds to a solution of this synthesis problem.

Refinement In Section 2.3.2, we observed that distinguishing between introductory forms I and elimination forms E helped us identify when a type should be treated as an input or output during typechecking. This insight also applies to how we treat examples in tandem with term generation. With introduction forms, the shape of the term and its corresponding type dictate how we refine example values. For example, at base type, our example values consist of constants. IREFINE-BASE says that we can synthesize a particular constant c_k if the examples all agree on that constant, *i.e.*, the goal example value of each example world is c_k. Note that if the example worlds contain differing constants, then this rule does not apply, and we would need to synthesize some other expression instead.

In contrast, at arrow type our example values consist of a set of partial functions. IREFINE-ARR says that we can synthesize a lambda if we are able to synthesize its body. To understand how IREFINE-ARR refines examples, first consider the case where X is a single example world, $\sigma \mapsto \overline{v_i \Rightarrow \chi_i}^{i<m}$. The example value is a single partial function consisting of m input/output examples. In this situation, the **apply** meta-function generates a collection of m new example worlds. The goal example value of each world is the right-hand side of each input/output example, χ_i. The environment of each new world consists of the original environment σ extended with a binding for the lambda's variable, x. The value bound to that variable is the left-hand side of each input/output example, v_i. We then use these new example worlds to synthesize the body of the lambda.

In general, X may contain n example worlds $\sigma_1 \mapsto \rho_1, \ldots, \sigma_n \mapsto \rho_n$. When there are multiple example worlds, we simply **apply** each of the example worlds to generate a collection $X_1, \ldots, X_n$ of example worlds and then concatenate them all together to create the final collection of example worlds X' to be used to synthesize the body of the lambda. This occurs if the user specifies multiple partial functions rather than a single partial function.

Surprisingly, there is a subtle semantic distinction between a collection of partial functions and a single partial function. To see this, consider synthesizing at goal type $T \to T$ with the examples:

$$\rho_1 = \sigma_1 \mapsto c_1 \Rightarrow c_2 \mid c_3 \Rightarrow c_4$$
$$\rho_2 = \sigma_2 \mapsto c_5 \Rightarrow c_6 \mid c_7 \Rightarrow c_8 \mid c_9 \Rightarrow c_{10}.$$

If we apply IREFINE-ARR, the resulting set of example worlds that we use to synthesize the body of lambda is:

$$X' = [x/c_1]\sigma_1 \mapsto c_2, [x/c_3]\sigma_1 \mapsto c_4, \\ [x/c_5]\sigma_2 \mapsto c_6, [x/c_7]\sigma_2 \mapsto c_8, [x/c_9]\sigma_2 \mapsto c_{10}.$$

In contrast, if we combined the two examples worlds into a single partial function,

$$\rho = \sigma \mapsto c_1 \Rightarrow c_2 \mid c_3 \Rightarrow c_4 \mid c_5 \Rightarrow c_6 \mid c_7 \Rightarrow c_8 \mid c_9 \Rightarrow c_{10},$$

then applying IREFINE-ARR results in the following examples,

$$X' = [x/c_1]\sigma \mapsto c_2, [x/c_3]\sigma \mapsto c_4, \\ [x/c_5]\sigma \mapsto c_6, [x/c_7]\sigma \mapsto c_8, [x/c_9]\sigma \mapsto c_{10}.$$

The only difference between the two cases is the original environments, σ_1 and σ_2 versus σ, that we extend when deriving X'. The initial environments shared among all the example worlds ought to be identical (as the values for the free variables of the program must be the same), so in our simply-typed setting, there is no practical difference between a collection of partial functions and a single partial function. However, in Chapter 5, we explore changes to IREFINE-ARR that make this fact no longer true.

Guessing In contrast, with E-terms, we are unable to refine the examples in this manner. This is because an E-term can be generated at any type and the shape of the E-term alone does not determine how we can decompose examples in a meaningful way. In particular, say that we try to synthesize a function application and choose a particular term E with some function type $\tau_1 \to \tau_2$. When synthesizing I, we would like to pass refined examples X' such that $E\ I$ satisfies X. However, this requires reasoning about the behavior of E, an arbitrary function, which is difficult to do. In the presence of richer language features, *e.g.*, recursion, this is undecidable.

Therefore, for E-terms, the EGUESS judgment resorts to term enumeration as developed in Section 2.1.1. The rules for generating variables (EGUESS-VAR) and application (EGUESS-APP) merely generate a term of an appropriate type. Note that when generating the argument to an application I in EGUESS-APP, we pass the *empty* set of examples to the IREFINE judgment. When X is empty, you can see that the IREFINE judgment degenerates to simple term enumeration.

However, we are not content generating any E-term of an appropriate type. We want such a term that satisfies the provided examples. The bridge rule IREFINE-GUESS enforces this by not only generating an E-term, but also explicitly ensuring that E satisfies X, written $I \vDash X$ (noting that Es are a proper subset of the Is). This satisfaction judgment ensures that, for each example world, evaluating I under

that environment results in a value v that is compatible to the goal example value for that world. Environment or substitution application, written $\sigma(I)$, produces an external language expression e suitable for evaluation.

This compatibility relation[3], written $v \simeq \chi$, only needs to compare a value v and some goal example value χ. At base type, v and χ are constants so it suffices to ensure they are the same constant (EQ-CTOR). At arrow type, v is a lambda and χ is a partial function. To ensure compatibility, it is sufficient to check for each input/output pair of the partial function that running v on the input of the pair produces a value compatible with the output of the pair (EQ-LAM-PF).

2.3.5 Synthesis Examples

To better understand how $\lambda^{\rightarrow}_{syn}$ operates, let's work through a number of examples.

Example 2.3.1. First, consider the degenerate case where the example set is empty ($X = \cdot$). We claimed in Section 2.3.4 that the IREFINE judgment degenerates to raw term enumeration when the example set is empty. To see that this is true, let's look at how each IREFINE rule behaves with no examples. At arrow type, IREFINE-ARR applies and passes along the empty example set ($X' = \cdot$) when synthesizing the body of the lambda. At base type, IREFINE-BASE applies and allows us to synthesize any constant c_k since X is empty. Finally, IREFINE-GUESS allows us to synthesize any E because $E \vDash \cdot$ holds vacuously.

Example 2.3.2. Consider the example $X = \cdot \mapsto c_1 \Rightarrow c_1 \mid c_2 \Rightarrow c_2$ with goal type $T \to T$ in the empty context. The following is a valid synthesis derivation of the identity function:

$$\dfrac{\dfrac{\dfrac{\dfrac{x{:}T \in x{:}T}{x{:}T \vdash T \rightsquigarrow x}\ \textsc{eguess-var} \quad \dfrac{[c_1/x](x) \longrightarrow^* c_1 \wedge [c_2/x](x) \longrightarrow^* c_2}{x \vDash [c_1/x] \mapsto c_1, [c_2/x] \mapsto c_2}\ \textsc{satisfies}}{x{:}T \vdash T \rhd [c_1/x] \mapsto c_1, [c_2/x] \mapsto c_2 \rightsquigarrow x}\ \textsc{irefine-guess}}{\cdot \vdash T \to T \rhd \cdot \mapsto c_1 \Rightarrow c_1 \mid c_2 \Rightarrow c_2 \rightsquigarrow \lambda x{:}T.\, x}\ \textsc{irefine-arr}$$

With our synthesis judgments, we always make a number of IREFINE derivations ending in a number of EGUESS derivations. This observation leads to a critical implementation optimization, *refinement trees*, that we explore in Chapter 7.

Example 2.3.3. Consider under-constraining the example set from the previous example: $X = c_1 \Rightarrow c_1$ with goal type $T \to T$ in the empty context. We can now

[3]Because the compatibility relation is between two different syntactic classes, it is not an equivalence as it cannot be reflexive, even though it tries to equate a value and an example value.

synthesize the constant function with a simpler derivation:

$$\dfrac{\dfrac{X = [c_1/x] \mapsto c_1}{x{:}T \vdash T \rhd [c_1/x] \mapsto c_1 \rightsquigarrow c_1}\ \textsc{irefine-base}}{\cdot \vdash T \to T \rhd \cdot \mapsto c_1 \Rightarrow c_1 \rightsquigarrow \lambda x{:}T.\, c_1}\ \textsc{irefine-arr}$$

Note that the identity function is still derivable in this context:

$$\dfrac{\dfrac{\dfrac{\textsc{eguess-var}\ \ x{:}T \in x{:}T}{x{:}T \vdash T \rightsquigarrow x} \qquad \dfrac{\textsc{satisfies}\ \ [c_1/x](x) \longrightarrow^* c_1}{x \vDash [c_1/x] \mapsto c_1}}{x{:}T \vdash T \rhd [c_1/x] \mapsto c_1 \rightsquigarrow x}\ \textsc{irefine-guess}}{\cdot \vdash T \to T \rhd \cdot \mapsto c_1 \Rightarrow c_1 \rightsquigarrow \lambda x{:}T.\, x}\ \textsc{irefine-arr}$$

How might we choose one program over the other? While both programs are the same size, we may choose the program with the smaller derivation tree (the constant function), or we might choose the program that uses the most variables (the identity function). With our synthesis relation, we are only concerned with ensuring that every satisfying program has a corresponding derivation in the system, so this ambiguity is irrelevant. However, in a synthesis procedure, we will need to disambiguate between these programs in some manner, a topic we revisit in Chapter 7.

Example 2.3.4. Next, consider synthesizing the if function from Section 2.2.2. Again we use the shorthand:

$$\mathsf{bool} \stackrel{\text{def}}{=} T \to T \to T$$
$$\mathsf{true} \stackrel{\text{def}}{=} \lambda t{:}T.\, \lambda f{:}T.\, t$$
$$\mathsf{false} \stackrel{\text{def}}{=} \lambda t{:}T.\, \lambda f{:}T.\, f.$$

Here, we are trying to synthesize a program of type $\mathsf{bool} \to T \to T \to T$ with the example set

$$X = \cdot \mapsto \mathsf{true} \Rightarrow c_1 \Rightarrow c_2 \Rightarrow c_1, \cdot \mapsto \mathsf{false} \Rightarrow c_1 \Rightarrow c_2 \Rightarrow c_2.$$

We can synthesize the desired if function in $\lambda^{\to}_{syn}$ as follows. First we can apply IREFINE-ARR three times to arrive at the synthesis state:

$$b{:}\mathsf{bool}, t{:}T, f{:}T \vdash T \rhd X \rightsquigarrow \blacksquare$$

where the example set X is:

$$[\mathsf{true}/b][c_1/t][c_2/f] \mapsto c_1$$
$$[\mathsf{false}/b][c_1/t][c_2/f] \mapsto c_2$$

Now, we apply IREFINE-GUESS to generate the application. The following EGUESS derivation generates the application itself (with $\Gamma = b{:}\mathsf{bool}, t{:}T, f{:}T$).

$$\dfrac{\dfrac{\dfrac{b{:}T \to T \to T \in \Gamma}{\Gamma \vdash T \to T \to T \rightsquigarrow b}\,\textsc{eguess-var} \qquad \dfrac{t{:}T \in \Gamma}{\Gamma \vdash T \rightsquigarrow t}\,\textsc{eguess-var}}{\Gamma \vdash T \to T \rightsquigarrow b\ t}\,\textsc{eguess-app} \qquad \dfrac{f{:}T \in \Gamma}{\Gamma \vdash T \rightsquigarrow f}\,\textsc{eguess-var}}{\Gamma \vdash T \rightsquigarrow b\ t\ f}\,\textsc{eguess-app}$$

Finally, we verify that $b\ t\ f$ satisfies the examples:

$$[\mathsf{true}/b][c_1/t][c_2/f](b\ t\ f) \longrightarrow^* c_1$$
$$[\mathsf{false}/b][c_1/t][c_2/f](b\ t\ f) \longrightarrow^* c_2.$$

Example 2.3.5. For our last example, consider what happens if we provide contradictory examples. Suppose that we are synthesizing a program of type $T \to T$ and provide the partial function $c_1 \Rightarrow c_1 \mid c_1 \Rightarrow c_2$. There is no function that we can synthesize that satisfies this specification as c_1 is mapped to two distinct outputs. To see this within $\lambda^{\to}_{syn}$, note that IREFINE-CTOR will never apply because the goal example values are different, and IREFINE-GUESS will never succeed because we will never be able to generate an E that satisfies both c_1 and c_2 given that the argument to the function is bound to c_1. Consequently there are no valid synthesis derivations in $\lambda^{\to}_{syn}$ for contradictory examples.

Detecting such contradictory examples turns out to be tricky and undecidable given a sufficiently rich language. At first glance, it seems like a simple syntactic check is sufficient: check to see that the partial functions involved map similar inputs to similar outputs. However, because we allow lambdas as inputs to partial functions, this problem reduces to deciding function equivalence. Furthermore, even this check is insufficient. For example, consider the worlds $[c_1/x] \mapsto c_1 \Rightarrow c_1, [c_2/x]c_1 \Rightarrow c_2$. While the partial functions map c_1 to distinct outputs, they still do not contradict each other because we fulfill the eventual goal example value with the variable x. Thus, we must decide not only equivalence between example values but also environments. Throughout this work, we assume that examples given by the user are non-contradictory, leaving detection to future work.

2.4 Related Work

In this chapter, we have demonstrated the fundamentals of program synthesis with types. With these fundamentals established, we now make technical comparisons of this program synthesis style with the prior approaches that we discussed in Section 1.1. In particular, previous work has explored program synthesis techniques using both types and examples as modes of specification, but not in the type-theoretic style that we propose. With our foundations, we can explain the salient features of these systems in terms of type theory and our own work.

2.4.1 Example Refinement

In $\lambda^{\rightarrow}_{syn}$, we make a distinction between the terms that allow example refinement—introduction terms—and the terms that do not—elimination forms. Ideally, we would allow example refinement through all the terms of our language. However, this is difficult to do for our elimination forms. In particular, recall $\lambda^{\rightarrow}_{syn}$'s rule for synthesizing function application:

$$\textsc{eguess-app}\ \frac{\Gamma \vdash \tau_1 \to \tau_2 \rightsquigarrow E \qquad \Gamma \vdash \tau_1 \rhd \cdot \rightsquigarrow I}{\Gamma \vdash \tau_2 \rightsquigarrow E\ I}$$

To refine examples through the function application $E\ I$, we may guess a particular function E, refine the examples in some manner, and feed them to the synthesis subproblem for I. The examples should be refined in such a way that we synthesize an I such that $E\ I$ satisfies the original set of examples. The problem is that, in general, this requires that we invert the behavior of E to discover how to refine the examples to satisfy this property. For arbitrary E, this is impossible to do as the function may not have an inverse (*i.e.*, it is surjective but not a bijection).

However, we are able to make progress if we are able to make assumptions about the possible set of functions that can appear in an application. These sorts of assumptions are made by several *combinator-based synthesis systems* such as λ^2 [Feser et al., 2015] and FlashExtract [Le and Gulwani, 2014] to push examples through the application of a fixed set of functions. For example, suppose that we are synthesizing a function with the single input/output example:

$$[0, 1, 2, 3] \Rightarrow [1, 2, 3, 4].$$

Then, because the behavior of the standard map function is

$$\begin{array}{l}\mathsf{let\ rec\ map}\ f\ l = \\ \quad \mathsf{match}\ l\ \mathsf{with} \\ \quad \mid\ [] \to [] \\ \quad \mid\ x :: xs \to f\ x :: \mathsf{map}\ f\ xs\end{array}$$

we know that we can synthesize a program of the form $\lambda l{:}\mathsf{list}.\,\mathsf{map}\,\blacksquare\,l$ where the hole is a function of type $\mathsf{nat} \to \mathsf{nat}$ with refined examples:

$$0 \Rightarrow 1 \;\mid\; 1 \Rightarrow 2 \;\mid\; 2 \Rightarrow 3 \;\mid\; 3 \Rightarrow 4.$$

Both λ^2 and FlashExtract admit variants of this rule for map and other functional combinators.

Such rules are admissible in our system. For example, here is an equivalent rule for I-refining map in our system extended with recursion (Chapter 5) and polymorphism (Chapter 9).

$$\textsc{irefine-map}\quad \frac{X = \overline{\sigma_i \mapsto [\chi_{1i}, \ldots, \chi_{ki}] \Rightarrow [\chi'_{1i}, \ldots, \chi'_{ki}]}^{i<n} \quad X' = \overline{\sigma_i \mapsto \chi_{1i} \Rightarrow \chi'_{1i}, \ldots, \sigma_i \mapsto \chi_{ki} \Rightarrow \chi'_{ki}}^{i<n} \quad \Gamma \vdash \alpha \to \beta \rhd X' \rightsquigarrow f}{\Gamma \vdash \alpha\,\mathsf{list} \to \beta\,\mathsf{list} \rhd X \rightsquigarrow \lambda l{:}\alpha\,\mathsf{list}.\,\mathsf{map}\,f\,l}$$

Intuitively, this rule synthesizes a function f for map by breaking up the input/output examples between lists point-wise, creating new examples that describe the required behavior of f. Our E-guessing rules provide a necessary fall back for when these shortcuts are not possible.

2.4.2 Forwards and Backwards Search

We can interpret synthesis in $\lambda^{\to}_{syn}$ as proof search through the Curry-Howard isomorphism [Howard, 1980] where program synthesis at a particular type is equivalent to proof search of a particular proposition. In this light, $\lambda^{\to}_{syn}$ specifies a *backwards search* for a satisfying program. That is, $\lambda^{\to}_{syn}$ exclusively refines the goal during the synthesis process. Other proof search-inspired also take advantage of backwards search, *e.g.*, InSynth [Gvero et al., 2013].

In contrast, we could instead perform a *forward search* by refining our context during synthesis. Consider extending our context so that it hold arbitrary expressions rather than just variables. Now, starting with our free variables, we can incrementally grow the context by enumerating well-typed terms in some order, for example, increasing size. At each step, we can check to see if we can construct a program from the *components* in the context that satisfies our examples. This style of forwards search-based program synthesis is performed by Escher [Albarghouthi et al., 2013] which creates a satisfying program (in a Lisp-like programming language) from a set of components built up from atoms.

There are benefits and drawbacks to employing either a backwards or forwards-based program search. Our backwards search allows us to decompose examples *during* goal refinement which allows us to greatly cut down the space of programs during synthesis. However, in the presence of other types such as products,

pure backwards search can be inefficient because it introduces unnecessary nondeterminism into the search process. To obtain the best of both worlds, we can adopt a search strategy that combines backwards and forwards by using *focusing*. Scherer and Rèmy [2015] use a proof search strategy equipped with focusing to efficiently find inhabitants of some given type, and we briefly explore the addition of focusing to $\lambda^{\rightarrow}_{syn}$ in conjunction with products in Section 4.1.1.

2.4.3 Type-directed Synthesis Systems

An alternative view of $\lambda^{\rightarrow}_{syn}$ is that it solves the *type inhabitation problem*—given a type, produce an inhabitant of that type. Several other systems have phrased the synthesis problem in terms of type inhabitation. For example, PROSPECTOR [Mandelin et al., 2005] auto-completes API queries by building up chains of object method calls called *jungloids*. The discovery of these jungloids is type-directed: given a pair of types (τ_1, τ_2) find a chain of jungloids whose composition has type $\tau_1 \rightarrow \tau_2$. It is analogous to repeated applications of our E-guessing rules, in particular EGUESS-APP which computes one such function in this chain.

Perelman et al. [2012] use types to provide auto-completion for the C# programming language. To do this, they generate well-typed program completions for a given completion query and rank those programs according to a number of heuristics including depth, name, namespace, and *type distance*—the distance between two types in the type hierarchy. INSYNTH [Gvero et al., 2013] performs similar sorts of auto-completion for the Scala programming language, but it draws its rankings or weights of components by analyzing corpora of code. These sorts of heuristics are also admissible in our system, similar to the example refinement heuristics discussed previously. In particular, these rankings could be adapted to the E-term generation algorithms we discuss in Chapter 7 to prioritize particular E-terms over others.

Finally, it is worthwhile to note that none of these type-directed systems that we have discussed include examples as part of their specification. Really, $\lambda^{\rightarrow}_{syn}$ solves the *type inhabitation and example satisfaction problem* where we are interested in finding programs of a particular type that satisfy the given examples. The example-refining techniques that we have developed here could be adapted to these systems, so that they can benefit from the additional specification power that examples provide.

Chapter 3

The Metatheory of $\lambda^{\rightarrow}_{syn}$

So far, we have introduced $\lambda^{\rightarrow}_{syn}$, a core calculus for program synthesis with types. $\lambda^{\rightarrow}_{syn}$ specifies a non-deterministic synthesis relation that requires modification to become an algorithm. However, in moving towards an algorithm, we would like to understand whether the design decisions we make to gain determinism, tractability, and expressiveness are admissible in our system or result in weakening the properties of $\lambda^{\rightarrow}_{syn}$.

In this section, we establish these properties by proving soundness and completeness of synthesis in $\lambda^{\rightarrow}_{syn}$. We present the full details of the proofs here to provide a template for reasoning about the metatheory of the more complicated type-directed program synthesis systems that we discuss in later chapters.

3.1 Soundness

Soundness states that the programs we synthesize are "correct". In type-directed program synthesis, correctness has two components:

- The synthesized program is well-typed at the goal type, *type soundness*, and
- The synthesized program satisfies the examples, *example soundness*.

First we show that type soundness holds for $\lambda^{\rightarrow}_{syn}$.

Lemma 3.1.1 (Type Soundness of $\lambda^{\rightarrow}_{syn}$).

1. *If* $\Gamma \vdash \tau \rightsquigarrow E$, *then* $\Gamma \vdash E \Rightarrow \tau$.
2. *If* $\Gamma \vdash X : \tau$ *and* $\Gamma \vdash \tau \rhd X \rightsquigarrow I$, *then* $\Gamma \vdash I \Leftarrow \tau$.

Proof. By mutual induction on the synthesis derivations for E- and I-terms. Consider the final rule used in the derivation:

Case EGUESS-VAR: $E = x$. By the premise of EGUESS-VAR, $x{:}\tau \in \Gamma$ which is sufficient to conclude by T-EVAR that x is well-typed.

Case EGUESS-APP: $E = E_1\ I$. By the premises of EGUESS-APP and our inductive hypotheses, we know that E_1 and I are well-typed at $\tau_1 \to \tau$ and τ_1, respectively. With this, we can conclude via T-EAPP that $E_1\ I$ is well-typed at type τ.

Case IREFINE-GUESS: $I = E$. By the premises of IREFINE-GUESS and our inductive hypothesis, we know that E is well-typed as an E-form and from T-IELIM E is well-typed as an I-form.

Case IREFINE-CTOR: $I = c$ and $\tau = T$. We can immediately conclude by T-ICTOR that c has type T.

Case IREFINE-ARR: $I = \lambda x{:}\tau_1.\, I_1$ and $\tau = \tau_1 \to \tau_2$. By the premises of IREFINE-ARR and our inductive hypothesis, we know that I_1 is well-typed at type τ_2. Therefore, we can conclude by T-ILAM that $\lambda x{:}\tau_1.\, I_1$ is well-typed at type $\tau_1 \to \tau_2$. □

Intuitively, type soundness follows trivially from the fact that we derived the synthesis algorithm from the type checking judgment. Because of this, we assume that type soundness holds for all of the systems we consider in this work and focus on example soundness, instead.

To prove example soundness, we need to show that each IREFINE rule (other than IREFINE-GUESS) produces well-typed example sets in order to invoke our inductive hypothesis. From there, we need to show that given satisfying subexpressions for the refined examples, our overall expression satisfies the original examples. These two steps manifest themselves into two lemmas that we must prove for each rule: *type preservation* and *satisfaction soundness*, respectively.

Lemma 3.1.2 (Type Preservation of apply). *If* $\Gamma \vdash \sigma \mapsto \overline{v_i \Rightarrow \chi_i}^{i<m} : \tau_1 \to \tau_2$ *then* $x{:}\tau_1, \Gamma \vdash \mathbf{apply}(x, \sigma, \overline{v_i \Rightarrow \chi_i}^{i<m}) : \tau_2$.

Proof. Unfolding the definition **apply**, we know that $\mathbf{apply}(x, \sigma_i, \overline{v_i \Rightarrow \chi_i}^{i<m}) = [v_1/x]\sigma \mapsto \chi_1, \ldots, [v_m/x]\sigma \mapsto \chi_m$. To show that the new example set has type τ_2, it suffices to show (through T-EXW-CONS) that:

- For all $i \in 1, \ldots, m$, $[v_i/x]\sigma$ is well-typed. We know that σ is well-typed because the original example set is well-typed. The additional binding $[v_i/x]$ is well-typed because the extended context demands that x has type τ_1 and v_i has type τ_1 because the partial function is well-typed.

- For all $i \in 1, \ldots, m$ and $y{:}\tau \in x{:}\tau_1, \Gamma$, there exists v such that $[y/v] \in [v_i/x]\sigma$. Because the original example set is well-typed, we know that σ covers all of the bindings in Γ. We know that the additional type binding for x is covered by the additional environment binding $[v_i/x]$.
- For all $i \in 1, \ldots, m$, χ_i has type τ_2. Because the partial function is well-typed, we know through T-EX-PF that each χ_i has type τ_2. □

Lemma 3.1.3 (Type Preservation of Example World Concatenation). *If* $\Gamma \vdash X : \tau$ *and* $\Gamma \vdash X' : \tau$ *then* $\Gamma \vdash X +\!\!+ X' : \tau$.

Proof. From T-EXW-CONS, we know that each individual example world in X and X' is well-typed at type τ by a straightforward induction on X and X', respectively. By T-EXW-CONS, we can cons together all of these example worlds together to form a single set of example worlds that is well-typed at type τ. □

Lemma 3.1.4 (Satisfaction Soundness of apply). *If* $I \vDash X'$ *then* $\lambda x{:}\tau.\, I \vDash \overline{\sigma_i \mapsto \rho_i}^{i<n}$ *where* $X' = \mathbf{apply}(x, \sigma_i \mapsto \rho_1) +\!\!+ \ldots +\!\!+ \mathbf{apply}(x, \sigma_n \mapsto \rho_n)$.

Proof. Consider a single example world $\sigma \mapsto \overline{v_i \Rightarrow \chi_i}^{i<m} \in X$. Unfolding the definition of the satisfies judgment for I shows that:

$$\sigma(I) = \sigma(\lambda x{:}\tau_1.\, I_1) \longrightarrow^* \lambda x{:}\tau_1.\, \sigma(I_1).$$

Therefore, it suffices to show that $\lambda x{:}\tau_1.\, \sigma(I_1) \simeq \overline{v_i \Rightarrow \chi_i}^{i<m}$. By EQ-LAM-PF, this means that we must show that for all $i \in 1, \ldots, m$,

$$(\lambda x{:}\tau_1.\, \sigma(I_1))\, v_i \longrightarrow [v_i/x]\sigma(I_1) \longrightarrow^* v \wedge v \simeq \chi_i$$

However, this follows directly from the fact that $I_1 \vDash X'$ where each example world in X' is of the form $[v_i/x]\sigma \mapsto \chi_i$. □

Lemma 3.1.2 and Lemma 3.1.3 allow us to conclude type preservation of example refinement at arrow types. Lemma 3.1.4 tells that **apply** manipulates its examples in a sound manner. With these two facts, we can now prove example soundness.

Lemma 3.1.5 (Example Soundness of $\lambda^{\rightarrow}_{syn}$). *If* $\Gamma \vdash X : \tau$ *and* $\Gamma \vdash \tau \rhd X \rightsquigarrow I$, *then* $I \vDash X$.

Proof. By induction on the synthesis derivation of I.

Case IREFINE-GUESS: $I = E$. Satisfaction of X follows directly from the premises of IREFINE-GUESS.

Case IREFINE-BASE: $I = c$ and $\tau = T$. By the premise of IREFINE-BASE, we know that $X = \sigma_1 \mapsto c, \ldots, \sigma_n \mapsto c$. We conclude satisfaction directly by noting that for all $\sigma \mapsto c \in X$, $\sigma(I) = \sigma(c) \longrightarrow^* c \simeq c$.

Case IREFINE-ARR: $I = \lambda x{:}\tau_1.\, I_1$ and $\tau = \tau_1 \to \tau_2$. To show that $I \vDash X$, we first must establish that the refined example set X' is satisfied by the body of the lambda I_1. By the premise of IREFINE-ARR, we know that

$$X = \sigma_1 \mapsto \rho_1, \ldots, \sigma_n \mapsto \rho_n$$
$$X' = \mathbf{apply}(x, \sigma_1, \rho_1) +\!\!+ \ldots +\!\!+ \mathbf{apply}(x, \sigma_n, \rho_n).$$

By Lemma 3.1.2, we know that each individual **apply** call produces a well-typed set of example worlds at type τ_2 and by Lemma 3.1.3, we know that X' is well-typed at τ_2. Therefore, we can invoke our inductive hypothesis to conclude that $I_1 \vDash X'$ and apply Lemma 3.1.4 to prove our goal. □

3.2 Completeness

Before we prove completeness we need several auxiliary theorems and lemmas. The first theorem is strong normalization which states that every term evaluates to a value. The satisfaction judgment (SATISFIES) relies on evaluation to values, so whenever we need to show that a program satisfies a set of examples, we implicitly rely on strong normalization to guarantee that evaluation produces a value.

Lemma 3.2.1 (Strong Normalization of $\lambda^{\to}_{syn}$). *If $\Gamma \vdash e : \tau$ then either e is a value or $e \longrightarrow^* v$.*

Proof. The external language fragment of $\lambda^{\to}_{syn}$ is merely $\lambda^{\to}$ (extended with a base type) which is known to be strongly normalizing [Tait, 1967]. □

Next, we need to reason about the shape of well-typed example values.

Lemma 3.2.2 (Example Value Canonicity). *If $\Gamma \vdash \chi : \tau$ then:*

1. *If $\tau = T$ then $\chi = c$.*
2. *If $\tau = \tau_1 \to \tau_2$ then $\chi = \rho$.*

Proof. By case analysis on τ and the well-typedness of χ, noting that there is a one-to-one correspondence between the shape of a type and an example value of that type. □

Finally, we need to show example refinement performed by the IREFINE rules preserve example satisfaction. Here is the relevant lemma for refining at arrow types.

Lemma 3.2.3 (Satisfaction Preservation of apply). *If* $\lambda x{:}\tau.\, I \vDash \overline{\sigma_i \mapsto \rho_i}^{i<n}$ *then* $I \vDash X'$ *where* $X' = \mathbf{apply}(x, \sigma_i \mapsto \rho_1) +\!\!+ \ldots +\!\!+ \mathbf{apply}(x, \sigma_n \mapsto \rho_n)$.

Proof. Consider a single example world $\sigma \mapsto \rho$. Unfolding the definition of SATISFIES for $\lambda x{:}\tau.\, I \vdash X$ and that example world:

$$\sigma(\lambda x{:}\tau_1.\, I) \longrightarrow \lambda x{:}\tau_1.\, \sigma(I) \wedge \lambda x{:}\tau_1.\, \sigma(I) \simeq \rho.$$

If $\rho = \overline{v_i \Rightarrow \chi_i}^{i<m}$, then by EQ-LAM-PF, we know that

$$\forall i \in 1, \ldots, n.\, (\lambda x{:}\tau_1.\, \sigma(I))\, v_i \longrightarrow [v_i/x]\sigma(I) \longrightarrow^* v \wedge v \simeq \chi_i.$$

By unfolding the definition of **apply** and $+\!\!+$, we know that X′ contains exactly every such $v_i \Rightarrow \chi_i$ as an example world $[v_i/x]\sigma \mapsto \chi_i$. Putting these two facts together, we can conclude that $I \vDash X'$. □

Now we are ready to tackle completeness. Intuitively, completeness states that we are able to synthesis all well-typed programs. First, we show the completeness of term enumeration in $\lambda^{\rightarrow}_{syn}$ which is straightforward.

Lemma 3.2.4 (Completeness of $\lambda^{\rightarrow}_{syn}$ Term Enumeration).

1. *If* $\Gamma \vdash E \Rightarrow \tau$, *then* $\Gamma \vdash \tau \rightsquigarrow E$.
2. *If* $\Gamma \vdash I \Leftarrow \tau$, *then* $\Gamma \vdash \tau \triangleright \cdot \rightsquigarrow I$.

Proof. By mutual induction on the typing derivations for E- and I-terms.

Case T-EVAR: $E = x$. By the premise of T-EVAR, $x{:}\tau \in \Gamma$ which is sufficient to conclude that $\Gamma \vdash \tau \rightsquigarrow x$ by EGUESS-VAR.

Case T-EAPP: $E = E_1\, I$. By the premises of T-EAPP, E_1 and I are well-typed, so we can invoke our inductive hypotheses to conclude that we can generate E_1 and I. By EGUESS-APP, we, therefore, can conclude that $\Gamma \vdash \tau \rightsquigarrow E_1\, I$.

Case T-IELIM: $I = E$. By T-IELIM and our inductive hypothesis, we know that $\Gamma \vdash \tau \rightsquigarrow E$. By IREFINE-GUESS, we can conclude that $\Gamma \vdash \tau \triangleright \cdot \rightsquigarrow E$ because $E \vdash \cdot$ holds trivially.

Case T-ICTOR: $I = c$ and $\tau = T$. By IREFINE-BASE, we can conclude that $\Gamma \vdash T \rhd \cdot \rightsquigarrow c$ because the premise of IREFINE-BASE holds trivially.

Case T-ILAM: $I = \lambda x{:}\tau_1.\, I_1$ and $\tau = \tau_1 \to \tau_2$. By T-ILAM, we know that I_1 is well-typed so we can invoke our inductive hypothesis to conclude that $x{:}\tau_1, \Gamma \vdash \tau_2 \rhd \cdot \rightsquigarrow I_1$. By IREFINE-ARR, we can conclude that $\Gamma \vdash \tau_1 \to \tau_2 \rhd \cdot \rightsquigarrow \lambda x{:}\tau_1.\, I_1$, noting that $X' = \cdot$ because $X = \cdot$. □

Like type soundness, term enumeration follows directly from the fact that our synthesis rules were derived from the type checking rules, so we focus on completeness of the system in the presence of examples.

One variant of synthesis completeness posits the existence of an example set that satisfies some well-typed program we would like to synthesis. However, it turns out this variant is trivially true!

Lemma 3.2.5 (Completeness of $\lambda^{\to}_{syn}$). *If $\Gamma \vdash I \Leftarrow \tau$ then there exists a X such if $\Gamma \vdash X : \tau$ and $I \vDash X$, then $\Gamma \vdash \tau \rhd X \rightsquigarrow I$.*

Proof. Consider the empty example set, $X = \cdot$. By Lemma 3.2.4, from X we are able to synthesize any well-typed I-term! □

The more interesting statement of completeness instead chooses a particular example set X. This statement makes the stronger claim that any example set satisfied by a program is sufficient to synthesize that program in $\lambda^{\to}_{syn}$.

Lemma 3.2.6 (Completeness of $\lambda^{\to}_{syn}$). *If $\Gamma \vdash I \Leftarrow \tau$, $\Gamma \vdash X : \tau$, and $I \vDash X$, then $\Gamma \vdash \tau \rhd X \rightsquigarrow I$.*

Proof. By induction on the typing derivation of I.

Case T-IELIM: $I = E$. By Lemma 3.2.4, $\Gamma \vdash \tau \rightsquigarrow E$. By IREFINE-GUESS and our premises (in particular, $E \vdash X$), we can conclude that $\Gamma \vdash \tau \rhd X \rightsquigarrow E$.

Case T-ICTOR: $I = c$ and $\tau = T$. Because X is well-typed at type T, we know by Lemma 3.2.2 that $X = \sigma_1 \mapsto c_1, \ldots, \sigma_n \mapsto c_n$. Furthermore, because $c \vdash X$, we know that for all $i \in 1, \ldots, n$, $\sigma_i(c) \longrightarrow c$ so $c \simeq c_i$. By EQ-CTOR, this means that each c_i is identical to c. Therefore, by IREFINE-BASE, we can conclude that $\Gamma \vdash T \rhd X \rightsquigarrow c$.

Case T-ILAM: $I = \lambda x{:}\tau_1.\, I_1$ and $\tau = \tau_1 \to \tau_2$. Because X is well-typed at $\tau_1 \to \tau_2$, we know by Lemma 3.2.2 that $X = \sigma_1 \mapsto \rho_1, \ldots, \sigma_n \mapsto \rho_n$. Let $X' = \mathbf{apply}(x, \sigma_1, \rho_1) +\!\!+ \ldots +\!\!+ \mathbf{apply}(x, \sigma_n, \rho_n)$. To invoke our inductive hypothesis to conclude that we can synthesize I_1, we must show that:

- $x{:}\tau_1, \Gamma \vdash I_1 \Leftarrow \tau_2$ which follows by the premise of T-ILAM.
- $x{:}\tau_1, \Gamma \vdash X' : \tau_2$ which follows by Lemma 3.1.2 and Lemma 3.1.3.
- $I_1 \vDash X'$ which follows by Lemma 3.2.3.

Therefore, we can invoke our inductive hypothesis to conclude that $x{:}\tau_1, \Gamma \vdash \tau_2 \rhd X' \rightsquigarrow I_1$. By IREFINE-ARR, we can therefore conclude that $\Gamma \vdash \tau_1 \to \tau_2 \rhd X \rightsquigarrow \lambda x{:}\tau_1.\, I_1$. □

In the above proofs, the bulk of the work resided in showing that our example refinement rules respected both type and example satisfaction. In summary, we need the following critical lemmas to prove soundness and completeness of synthesis in the presence of new language features:

- *Type preservation* lemmas stating that refined examples produced by the IREFINE rules are well-typed.
- *Satisfaction soundness* lemmas stating that satisfying sub-expressions can be put together to form a complete expression that satisfies the original examples.
- *Satisfaction preservation* lemmas stating that a satisfying expression implies that its component expressions satisfy the examples produced by the IREFINE rules.

Chapter 4

Simple Type Extensions

$\lambda^{\rightarrow}_{syn}$ is a core calculus for program synthesis with types. As such, it only contains the bare essence of a typed, functional programming language, namely lambdas and function application. However, one of the appeals of the type-directed synthesis approach is that we derive the synthesis judgment directly from the type checking judgment. In other words, type checking immediately gives us insight into program synthesis for new types!

In this chapter, we explore the process of integrating new types into $\lambda^{\rightarrow}_{syn}$ by considering a number of additional basic types: products, sums, and records. By playing this game with the type system of flipping inputs and outputs, we learn how to generate terms and refine examples of these types. In some cases, this is sufficient to synthesize this type within $\lambda^{\rightarrow}_{syn}$ immediately. However in other cases, in particular, the more complex types that we consider in future chapters, we must do additional work to properly synthesize programs of these types.

4.1 Products

Figure 4.1 gives the modifications to $\lambda^{\rightarrow}_{syn}$ necessary to add products to the system. The product type $\tau_1 \times \tau_2$ has an introduction form, the pair (e_1, e_2), and two elimination forms, left projection $\mathsf{fst}\, e$ and right projection $\mathsf{snd}\, e$. We add both forms to the external language e of expressions, and we add the pair introduction form (I_1, I_2) to the I grammar and the projection elimination forms $\mathsf{fst}\, E$ and $\mathsf{snd}\, E$ to the E grammar. This introduction form also serves as the value of the pair type (v_1, v_2) as well as its example value (χ_1, χ_2).

Type checking a pair amounts to type checking its components (T-PAIR and T-IPAIR). Type checking a projection results in the left- or right-hand side of the product type (T-FST and T-EFST, T-SND and T-ESND). We enforce a left-to-right ordering of evaluation with our evaluation contexts consistent with our choice of call-by-value evaluation order. Finally, checking that two pairs are compatible decomposes to checking that their components are compatible (EQ-PAIR).

$$
\begin{array}{lcll}
\tau & ::= & \ldots \mid \tau_1 \times \tau_2 & \text{Types} \\
e & ::= & \ldots \mid \mathsf{fst}\, e \mid \mathsf{snd}\, e \mid (e_1, e_2) & \text{Terms} \\
v & ::= & \ldots \mid (v_1, v_2) & \text{Values} \\
\mathcal{E} & ::= & \ldots \mid (\mathcal{E}, e) \mid (v, \mathcal{E}) & \text{Evaluation Contexts} \\
E & ::= & \ldots \mid \mathsf{fst}\, E \mid \mathsf{snd}\, E & \text{Elimination Terms} \\
I & ::= & \ldots \mid (I_1, I_2) & \text{Introduction Terms} \\
\chi & ::= & \ldots \mid (\chi_1, \chi_2) & \text{Example Values}
\end{array}
$$

$\boxed{\Gamma \vdash e : \tau}$

$$\textsc{t-pair}\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \textsc{t-fst}\ \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \mathsf{fst}\, e : \tau_1} \qquad \textsc{t-snd}\ \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \mathsf{snd}\, e : \tau_2}$$

$\boxed{\Gamma \vdash E \Rightarrow \tau}$

$$\textsc{t-Efst}\ \frac{\Gamma \vdash E \Rightarrow \tau_1 \times \tau_2}{\Gamma \vdash \mathsf{fst}\, E \Rightarrow \tau_1} \qquad \textsc{t-Esnd}\ \frac{\Gamma \vdash E \Rightarrow \tau_1 \times \tau_2}{\Gamma \vdash \mathsf{snd}\, E \Rightarrow \tau_2}$$

$\boxed{\Gamma \vdash I \Leftarrow \tau}$

$$\textsc{t-Ipair}\ \frac{\Gamma \vdash I_1 \Leftarrow \tau_1 \qquad \Gamma \vdash I_2 \Leftarrow \tau_2}{\Gamma \vdash (I_1, I_2) \Leftarrow \tau_1 \times \tau_2}$$

$\boxed{e \longrightarrow e'}$

$$\textsc{eval-fst}\ \mathsf{fst}\,(v_1, v_2) \longrightarrow v_1 \qquad \textsc{eval-snd}\ \mathsf{snd}\,(v_1, v_2) \longrightarrow v_2$$

$\boxed{\Gamma \vdash \tau \rightsquigarrow E}$

$$\textsc{eguess-fst}\ \frac{\Gamma \vdash \tau_1 \times \tau_2 \rightsquigarrow E}{\Gamma \vdash \tau_1 \rightsquigarrow \mathsf{fst}\, E} \qquad \textsc{eguess-snd}\ \frac{\Gamma \vdash \tau_1 \times \tau_2 \rightsquigarrow E}{\Gamma \vdash \tau_2 \rightsquigarrow \mathsf{snd}\, E}$$

$\boxed{\Gamma \vdash \tau \rhd X \rightsquigarrow I}$

$$\textsc{irefine-prod}\ \frac{X = \overline{\sigma_i \mapsto (v_{1i}, v_{2i})}^{i<n} \qquad \mathbf{proj}(X) = (X_1, X_2) \qquad \Gamma \vdash \tau_1 \rhd X_1 \rightsquigarrow I_1 \qquad \Gamma \vdash \tau_2 \rhd X_2 \rightsquigarrow I_2}{\Gamma \vdash \tau_1 \times \tau_2 \rhd X \rightsquigarrow (I_1, I_2)}$$

$\boxed{\Gamma \vdash \chi : \tau}$

$$\textsc{t-ex-pair}\ \frac{\Gamma \vdash \chi_1 : \tau_1 \qquad \Gamma \vdash \chi_2 : \tau_2}{\Gamma \vdash (\chi_1, \chi_2) : \tau_1 \times \tau_2}$$

$\boxed{v \simeq \chi}$

$$\textsc{eq-pair}\ \frac{v_1 \simeq \chi_1 \qquad v_2 \simeq \chi_2}{(v_1, v_2) \simeq (\chi_1, \chi_2)}$$

$$\mathbf{proj}(\overline{\sigma_i \mapsto (v_{1i}, v_{2i})}^{i<n}) = (X_1, X_2) \quad \text{where } X_1 = \overline{\sigma_i \mapsto v_{1i}}^{i<n}, X_2 = \overline{\sigma_i \mapsto v_{2i}}^{i<n}$$

Figure 4.1: $\lambda^{\rightarrow}_{syn}$ products

We derive the synthesis rules for projections (EGUESS-FST and EGUESS-SND) and pairs (IREFINE-PROD) directly from their typing rules. To synthesize a left-projection, $\mathsf{fst}\, E$, or a right-projection, $\mathsf{snd}\, E$, it is sufficient to synthesize a pair E of the appropriate product type. To refine a pair (T-EX-PAIR), we note that if our examples are well-typed at a product type, then they must be all be pairs. We extract the left-hand components of the example pairs and their corresponding environments. These form the example worlds that we use to synthesize the left-hand component of the pair I_1. The right-hand components and their corresponding environments become the example worlds that we use to synthesize the right-hand component, I_2. As a concrete example, consider the following example worlds:

$$X = \sigma_1 \mapsto (c_1, c_2), \sigma_2 \mapsto (c_3, c_4).$$

Then the two example worlds that we create in IREFINE-PROD are

$$X_1 = \sigma_1 \mapsto c_1, \sigma_2 \mapsto c_3$$
$$X_2 = \sigma_1 \mapsto c_3, \sigma_2 \mapsto c_4.$$

To prove soundness, we need type preservation and satisfaction soundness lemmas as discussed in Chapter 3. We can easily prove the necessary lemmas here:

Lemma 4.1.1 (Type Preservation of proj). *If* $\Gamma \vdash X : \tau_1 \times \tau_2$ *then* $\Gamma \vdash X_1 : \tau_1$ *and* $\Gamma \vdash X_2 : \tau_2$ *where* $\mathbf{proj}(X) = (X_1, X_2)$.

Proof. Immediate from the premise of the statement. T-EXW-CONS says that each σ_i is well-typed and T-EX-PAIR says that the example pairs are well-typed and their components are well-typed at τ_1 and τ_2. □

Lemma 4.1.2 (Satisfaction Soundness of proj). *If* $I_1 \vDash X_1$ *and* $I_2 \vDash X_2$ *then* $(I_1, I_2) \vDash X$ *where* $\mathbf{proj}(X) = (X_1, X_2)$.

Proof. Consider a single example world $\sigma \mapsto (\chi_1, \chi_2) \in X$. Unfolding the definition of SATISFIES for (I_1, I_2) shows that

$$\sigma(I) = \sigma(I_1, I_2) \longrightarrow^* (\sigma(I_1), \sigma(I_2))$$

Therefore, it suffices to show that $(\sigma(I_1), \sigma(I_2)) \simeq (\chi_1, \chi_2)$, and by EQ-PAIR, this means that we must show that $\sigma(I_1) \simeq \chi_1$ and $\sigma(I_2) \simeq \chi_2$. By the definition of proj, $\sigma \mapsto \chi_1 \in X_1$ and $\sigma \mapsto \chi_2 \in X_2$, so we know that by the definition of SATISFIES and the fact that $I_1 \vDash X_1$ and $I_2 \vDash X_2$ that $\sigma(I_1) \simeq \chi_1$ and $\sigma(I_2) \simeq \chi_2$. □

Completeness requires that we show that IREFINE-PROD preserves satisfaction of examples. The version of the lemma for **proj** is also straightforward to prove:

Lemma 4.1.3 (Satisfaction Preservation of proj). *If* $(I_1, I_2) \vDash X$ *then* $I_1 \vDash X_1$ *and* $I_2 \vDash X_2$ *where* $\mathbf{proj}(X) = (X_1, X_2)$.

Proof. Consider a single example world $\sigma \mapsto (\chi_1, \chi_2)$; the shape of the examples is guaranteed by example value canonicity. By SATISFIES, we know that $\sigma(I_1) \simeq \chi_1$ and $\sigma(I_2) \simeq \chi_2$. By unfolding **proj**, we know that this covers each of the example worlds in X_1 and X_2, so this is sufficient to conclude that $I_1 \vDash X_1$ and $I_2 \vDash X_2$. □

To show how we apply these lemmas, let's walk through the additional cases for proving example soundness and completeness in $\lambda^{\rightarrow}_{syn}$ extended with products.

Lemma 4.1.4 (Example Soundness of $\lambda^{\rightarrow}_{syn}$ with Products). *If* $\Gamma \vdash X : \tau$ *and* $\Gamma \vdash \tau \rhd X \rightsquigarrow I$, *then* $I \vDash X$.

Proof. **Case** IREFINE-PROD: $I = (I_1, I_2)$, $\tau = \tau_1 \times \tau_2$, with $\mathbf{proj}(X) = (X_1, X_2)$. By Lemma 4.1.1, $\Gamma \vdash X_1 : \tau_1$ and $\Gamma \vdash X_2 : \tau_2$. Therefore, by our inductive hypothesis, we can conclude that $I_1 \vDash X_1$ and $I_2 \vDash X_2$ and by Lemma 4.1.2 we can conclude our goal. □

Lemma 4.1.5 (Completeness of $\lambda^{\rightarrow}_{syn}$ with Products). *If* $\Gamma \vdash I \Leftarrow \tau$, $\Gamma \vdash X : \tau$, *and* $I \vDash X$, *then* $\Gamma \vdash \tau \rhd X \rightsquigarrow I$.

Proof. **Case** T-PROD: $I = (I_1, I_2)$ and $\tau = \tau_1 \times \tau_2$. Because X is well-typed at $\tau_1 \times \tau_2$, we know by Lemma 3.2.2 that $X = \overline{\sigma_i \mapsto (X_{1i}, X_{2i})}^{i<n}$. Let $(X_1, X_2) = \mathbf{proj}(X)$. To invoke our inductive hypothesis to conclude that we can synthesize (I_1, I_2), we must show that:

- $\Gamma \vdash I_1 \Leftarrow \tau_1$ and $\Gamma \vdash I_2 \Leftarrow \tau_2$ which follows by the premise of T-IPAIR.
- $\Gamma \vdash X_1 : \tau_1$ and $\Gamma \vdash X_2 : \tau_2$ which follows by Lemma 4.1.1.
- $I_1 \vDash X_1$ and $I_2 \vDash X_2$ which follows by Lemma 4.1.3.

Therefore, we can invoke our inductive hypothesis to conclude that $\Gamma \vdash \tau_1 \rhd X_1 \rightsquigarrow I_1$ and $\Gamma \vdash \tau_2 \rhd X_2 \rightsquigarrow I_2$. By IREFINE-PROD, we can therefore conclude that $\Gamma \vdash \tau_1 \times \tau_2 \rhd X \rightsquigarrow (I_1, I_2)$. □

With the appropriate lemmas, the proof of soundness and completeness for products is identical to the analogous proof for functions (modulo the lemmas). This reasoning template holds for the rest of the language extensions we consider in this chapter. Therefore, for the remaining types, we merely provide the critical lemmas for proving soundness and completeness.

4.1.1 Efficiency

Tuples bring up an interesting matter of efficiency. In order to use a tuple, we must project out either its left or right component. From the perspective of an implementation, this is undesirable because we must speculatively explore the derivations where we project out either component with fst and snd. These derivations may not be fruitful and become wasted work. Furthermore, if we require both components, then the order in which we apply fst and snd matters insofar as they represent two distinct-yet-equivalent derivations.

To alleviate these problems we import the technique of *focusing* from the proof search literature [Liang and Miller, 2007] into program synthesis. Here, we briefly describe the focusing procedure.[1] Intuitively, whenever we introduce a value of tuple type into the context, we greedily decompose the tuple down to base or arrow types. For example, suppose we have a variable x of type $T_1 \times ((T_2 \times T_3) \times T_4)$. Then by focusing on x, we obtain the following expressions of base type:

$$\begin{aligned} \mathsf{fst}\ x &: T_1 \\ \mathsf{fst}\ \mathsf{fst}\ \mathsf{snd}\ x &: T_2 \\ \mathsf{snd}\ \mathsf{fst}\ \mathsf{snd}\ x &: T_3 \\ \mathsf{snd}\ \mathsf{snd}\ x &: T_4 \end{aligned}$$

We extend our context Γ to not contain just free variables, but arbitrary E-terms. After focusing, we add these expressions to our context, making them available for use later in the synthesis derivation.

Essentially, focusing allows us to rip out all of the components of a product for use immediately in our program. This has the cost of increasing the size of our context which in turn affects the cost of raw E-term generation. But with focusing, we are able to consider programs involving projections much earlier in the synthesis process as well as avoid additional derivations involving projections. For example, suppose we are generating E-terms of type nat and have a variable x of type $\mathsf{nat} \times \mathsf{nat}$ in the context. Focusing brings fst x and snd x into the context for us to use. If we are exploring these derivations in terms of increasing size (as suggested in Section 2.1), then we can immediately use the projections of x rather than explore derivations of greater size to generate expressions involving fst and snd. If we need to use these projections in the final program, than this is a net win. However, if we do not need these projections, then we will needlessly generate E-terms involving them earlier in the synthesis process than we would without focusing.

[1] For a full treatment of focusing in program synthesis to efficiently handle tuples, see Frankle [2015].

4.2 Records

We can easily take the machinery that we derived for products and lift it into records. Figure 4.2 gives the syntax and semantics of records in $\lambda^{\rightarrow}_{syn}$. The record literal $\{\overline{l_i = I_i}^{i<m}\}$ introduces values of the record type $\{\overline{l_i{:}\tau_i}^{i<m}\}$ and record projection $l.E$ eliminates those values according to EVAL-RPROJ.

Naturally, the example value of a record type is the record value $\{\overline{l_i = \chi_i}^{i<m}\}$. Generating a projection is straightforward: guess a record expression that has a field of the goal type that you are after and project out that field (EGUESS-RPROJ). To make this efficient in practice, we can use focusing as we did with tuples to make available all of the possible projections up front rather than speculatively guessing them. Record I-refinement (IREFINE-RECORD) is identical to product I-refinement save for the presence of labels which are handled easily because they are recorded in the goal type. We know that the example values are record examples assuming that the examples are well-typed. Therefore, when synthesizing a field of a record, we use the collected examples values for that field during synthesis. The **rproj** meta-function performs this collection, generalizing the behavior of **proj** for pairs to m-ary records.

Consequently, the necessary lemmas to verify soundness and completeness are similar to the product case.

Lemma 4.2.1 (Type Preservation of rproj). *If* $\Gamma \vdash X : \{\overline{l_i{:}\tau_i}^{i<m}\}$ *then* $\overline{\Gamma \vdash X_i : \tau_i}^{i<m}$ *where* $\mathbf{rproj}(X) = X_1, \ldots, X_m$.

Proof. Similar to **proj**, the necessary conditions are immediate from the premise. T-EXW-CONS says that each σ_i is well-typed and T-EX-RECORD says that the example records are well-typed and their components are well-typed at $\tau_1, \ldots, \tau_m$. □

Lemma 4.2.2 (Satisfaction Soundness of rproj). *If* $\overline{I_i \vDash X_i}^{i<m}$ *then* $\{\overline{l_i = I_i}^{i<m}\}) \vDash X$ *where* $\mathbf{rproj}(X) = X_1, \ldots, X_m$.

Proof. Consider a single example world $\sigma \mapsto \{\overline{l_i = \chi_i}^{i<m}\} \in X$. By SATISFIES, we know that we must show that $\sigma(I_i) \longrightarrow^* v_i$ and $v_i \simeq \chi_i$ for each $i \in 1, \ldots, m$. However, by unfolding the definition of **proj** we see that each χ_i is distributed to X_i with the environment σ, and we know from our premise that his example world is satisfied by I_i. □

Lemma 4.2.3 (Satisfaction Preservation of rproj). *If* $\{\overline{l_i = I_i}^{i<m}\}) \vDash X$ *then* $\overline{I_i \vDash X_i}^{i<m}$ *where* $\mathbf{rproj}(X) = X_1, \ldots, X_m$.

$$
\begin{array}{lcll}
\tau & ::= & \ldots \mid \{\overline{l_i{:}\tau_i}^{\,i<m}\} & \text{Types} \\
e & ::= & \ldots \mid \{\overline{l_i = e_i}^{\,i<m}\} \mid e.l & \text{Terms} \\
v & ::= & \ldots \mid \{\overline{l_i = v_i}^{\,i<m}\} & \text{Values} \\
\mathcal{E} & ::= & \ldots \mid \{l_1 = v_1, \ldots, l = \mathcal{E}, \ldots, l_m = e_m\} \mid \mathcal{E}.l & \text{Evaluation Contexts} \\
E & ::= & \ldots \mid E.l & \text{Elimination Forms} \\
I & ::= & \ldots \mid \{\overline{l_i = I_i}^{\,i<m}\} & \text{Introduction Forms} \\
\chi & ::= & \ldots \mid \{\overline{l_i = \chi_i}^{\,i<m}\} & \text{Example Values}
\end{array}
$$

$\boxed{\Gamma \vdash e : \tau}$

$$\textsc{t-record}\ \frac{\overline{\Gamma \vdash e_i : \tau_i}^{\,i<m}}{\Gamma \vdash \{\overline{l_i = e_i}^{\,i<m}\} : \{\overline{l_i{:}\tau_i}^{\,i<m}\}} \qquad \textsc{t-rproj}\ \frac{\Gamma \vdash e : \{\overline{l_i{:}\tau_i}^{\,i<m}\}}{\Gamma \vdash e.l_i : \tau_i}$$

$\boxed{\Gamma \vdash I \Leftarrow \tau}$

$$\textsc{t-Irecord}\ \frac{\overline{\Gamma \vdash I_i \Leftarrow \tau_i}^{\,i<m}}{\Gamma \vdash \{\overline{l_i = I_i}^{\,i<m}\} : \{\overline{l_i{:}\tau_i}^{\,i<m}\}}$$

$\boxed{\Gamma \vdash \chi : \tau}$

$$\textsc{t-ex-record}\ \frac{\overline{\Gamma \vdash \chi_i : \tau_i}^{\,i<m}}{\Gamma \vdash \{\overline{l_i = \chi_i}^{\,i<m}\} : \{\overline{l_i{:}\tau_i}^{\,i<m}\}}$$

$\boxed{e \longrightarrow e'}$

$$\textsc{eval-rproj}\ \{\overline{l_i = v_i}^{\,i<m}\}.l_i \longrightarrow v_i$$

$\boxed{\Gamma \vdash \tau \rightsquigarrow E}$

$$\textsc{eguess-rproj}\ \frac{\Gamma \vdash \{\overline{l_i{:}\tau_i}^{\,i<m}\} \rightsquigarrow E}{\Gamma \vdash \tau_i \rightsquigarrow E.l_i}$$

$\boxed{v \simeq \chi}$

$$\textsc{eq-record}\ \frac{\overline{v_i \simeq \chi_i}^{\,i<m}}{\overline{l_i = v_i}^{\,i<m} \simeq \overline{l_i = \chi_i}^{\,i<m}}$$

$\boxed{\Gamma \vdash \tau \rhd X \rightsquigarrow I}$

$$\textsc{irefine-record}\ \frac{X = \overline{\sigma_j \mapsto \{l_1 = \chi_{1j}, \ldots, l_m = \chi_{mj}\}}^{\,j<n} \quad \mathbf{rproj}(X) = X_1, \ldots, X_m \quad \overline{\Gamma \vdash \tau_i \rhd X_i \rightsquigarrow I_i}^{\,i<m}}{\Gamma \vdash \{\overline{l_i{:}\tau_i}^{\,i<m}\} \rhd X \rightsquigarrow \{\overline{l_i = I_i}^{\,i<m}\}}$$

$$\mathbf{rproj}(\overline{\sigma_j \mapsto \{l_1 = \chi_{1j}, \ldots, l_m = \chi_{mj}\}}^{\,j<n}) = X_1, \ldots, X_m$$
where
$$\forall i \in 1, \ldots, m.\ X_i = \overline{\sigma_j \mapsto \chi_{ij}}^{\,i<m}$$

Figure 4.2: $\lambda^{\rightarrow}_{syn}$ records definitions

Proof. Consider a single example world $\sigma \mapsto \{\overline{l_i = \chi_i}^{i<m}\} \in X$. By SATISFIES, we know that $\overline{\sigma(I_i) \simeq \chi_i}^{i<m}$. By unfolding **rproj**, we know that this covers each of the example worlds among the $X_1, \ldots, X_m$, so this is sufficient to conclude that $I_1 \vDash X_1, \ldots, I_m \vDash X_m$. □

4.2.1 Subtyping and Synthesis

These rules do not introduce subtyping and the usual record rules for subtypes—exchange, record width, and record depth subtyping. While we do not give a full treatment of subtyping, it is worthwhile to briefly consider how we would add it to $\lambda^{\rightarrow}_{syn}$. We can certainly play the types-to-synthesis game with the usual subsumption rule,

$$\frac{\Gamma \vdash e : \tau' \qquad \tau' <: \tau}{\Gamma \vdash e : \tau},$$

to produce synthesis subsumption rules for E and I terms:

$$\frac{\Gamma \vdash I \rhd X \rightsquigarrow \tau' \qquad \tau' <: \tau}{\Gamma \vdash I \rhd X \rightsquigarrow \tau} \qquad \frac{\Gamma \vdash E \rightsquigarrow \tau' \qquad \tau' <: \tau}{\Gamma \vdash E \rightsquigarrow \tau}.$$

In our non-deterministic system, both rules are perfectly admissible. Intuitively, they state that rather than synthesizing at a particular goal type, we can synthesize at any subtype. This justifies having a subsumption rule for both E and I terms; would like to apply this logic to any goal type that admits subtyping.

Algorithmically this poses a great difficulty in that we now need to efficiently and completely search the dimension of subtypes in addition to the dimension of terms. For example, when E-guessing we may be able to satisfying a goal of record type by synthesizing a record that is a width subtype of the goal, *i.e.*, it has additional fields. To get to this point, however, we must grow our goal type so that we eventually search for terms of this (larger) subtype. One can imagine using an iterative deepening approach on the number subtype derivations similar to how we can enumerate terms in order of increasing size, but depending on the types involved, the number of subtypes may grow very quickly with the size of the subtyping derivation. Further implementation techniques might exploit the behavior of particular subtyping rules that we introduce into the system or heuristics to either prioritize likely subtypes or prune away unlikely or undesirable subtypes, sacrificing some completeness for tractability.

$$
\begin{array}{lcll}
\tau & ::= & \ldots \mid \tau_1 + \tau_2 & \text{Types} \\
e & ::= & \ldots \mid \mathsf{match}\ e\ \mathsf{with}\ \mathsf{inl}\, x_1 \to e_1 \mid \mathsf{inr}\, x_2 \to e_2 \mid \mathsf{inl}\, e \mid \mathsf{inr}\, e & \text{Terms} \\
v & ::= & \ldots \mid \mathsf{inl}\, v \mid \mathsf{inr}\, v & \text{Values} \\
\mathcal{E} & ::= & \ldots \mid \mathsf{match}\ \mathcal{E}\ \mathsf{with}\ \mathsf{inl}\, x_1 \to e_1 \mid \mathsf{inr}\, x_2 \to e_2 & \text{Eval Ctx} \\
E & ::= & \ldots & \text{Elim} \\
I & ::= & \ldots \mid \mathsf{match}\ E\ \mathsf{with}\ \mathsf{inl}\, x_1 \to I_1 \mid \mathsf{inr}\, x_2 \to I_2 \mid \mathsf{inl}\, I \mid \mathsf{inr}\, I & \text{Intros} \\
\chi & ::= & \ldots \mid \mathsf{inl}\, \chi \mid \mathsf{inr}\, \chi & \text{Ex. Values}
\end{array}
$$

$\boxed{\Gamma \vdash e : \tau}$

$$\textsc{t-inl}\ \frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathsf{inl}\, e : \tau_1 + \tau_2} \qquad \textsc{t-inr}\ \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathsf{inr}\, e : \tau_1 + \tau_2}$$

$$\textsc{t-match}\ \frac{\Gamma \vdash e : \tau_1 + \tau_2 \qquad x_1{:}\tau_1, \Gamma \vdash e_1 : \tau \qquad x_2{:}\tau_2, \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathsf{match}\ e\ \mathsf{with}\ \mathsf{inl}\, x_1 \to e_1 \mid \mathsf{inr}\, x_2 \to e_2 : \tau}$$

$\boxed{\Gamma \vdash I \Leftarrow \tau}$

$$\textsc{t-Iinl}\ \frac{\Gamma \vdash I \Leftarrow \tau_1}{\Gamma \vdash \mathsf{inl}\, I \Leftarrow \tau_1 + \tau_2} \qquad \textsc{t-Iinr}\ \frac{\Gamma \vdash I \Leftarrow \tau_2}{\Gamma \vdash \mathsf{inr}\, I \Leftarrow \tau_1 + \tau_2}$$

$$\textsc{t-Imatch}\ \frac{\Gamma \vdash E \Rightarrow \tau_1 + \tau_2 \qquad x_1{:}\tau_1, \Gamma \vdash I_1 \Leftarrow \tau \qquad x_2{:}\tau_2, \Gamma \vdash I_2 \Leftarrow \tau}{\Gamma \vdash \mathsf{match}\ E\ \mathsf{with}\ \mathsf{inl}\, x_1 \to I_1 \mid \mathsf{inr}\, x_2 \to I_2 : \tau}$$

$\boxed{\Gamma \vdash \chi : \tau}$

$$\textsc{t-ex-inl}\ \frac{\Gamma \vdash \chi : \tau_1}{\Gamma \vdash \mathsf{inl}\, \chi : \tau_1 + \tau_2} \qquad \textsc{t-ex-inr}\ \frac{\Gamma \vdash \chi : \tau_2}{\Gamma \vdash \mathsf{inr}\, \chi \Leftarrow \tau_1 + \tau_2}$$

$\boxed{e \longrightarrow e'}$

$$\textsc{eval-match-inl}\quad \mathsf{match}\ \mathsf{inl}\, v\ \mathsf{with}\ \mathsf{inl}\, x_1 \to e_1 \mid \mathsf{inr}\, x_2 \to e_2 \longrightarrow [v/x_1]e_1$$

$$\textsc{eval-match-inr}\quad \mathsf{match}\ \mathsf{inr}\, v\ \mathsf{with}\ \mathsf{inl}\, x_1 \to e_1 \mid \mathsf{inr}\, x_2 \to e_2 \longrightarrow [v/x_2]e_2$$

Figure 4.3: $\lambda^{\to}_{syn}$ sums definitions

4.3 Sums

Before adding algebraic data types (which we consider in Chapter 5), let's consider how we might add sums to $\lambda^{\rightarrow}_{syn}$. Figure 4.3 gives the syntax, type checking, and evaluation rules for sums. We introduce the sum type $\tau_1 + \tau_2$ with the constructors $\mathsf{inl}\, e$ and $\mathsf{inr}\, e$ which inject a value of type τ_1 and τ_2, respectively, into the sum. Note the fact that inl and inr are I-forms makes it evident that we do not need to provide a type annotation stating the sum types they belong to. Because they are I-forms, they are always checked against a sum type which renders the annotation unnecessary.[2]

Sum types are eliminated via pattern matching, written:

$$\mathsf{match}\ e\ \mathsf{with}\ \mathsf{inl}\, x_1 \to e_1 \mid \mathsf{inr}\, x_2 \to e_2$$

which performs case analysis on a particular sum value to see which constructor created it. EVAL-MATCH-INL and EVAL-MATCH-INR describes what happens when we have either an inl or inr in the *scrutinee position* of the sum. In either situation, we choose the appropriate branch of the pattern match, bind the value injected by the sum to a variable, and produce the corresponding expression of that branch.

Perhaps surprisingly, pattern matching is an I form rather than an E form even though it eliminates sums! This is because the branches of the pattern match act as binders, similarly to the body of a lambda. They do not directly participate in the reduction of the match and thus can be any (normal-form) expression. The result of the pattern match is the (shared) type of these branches—note that T-IMATCH says that the result type is some τ which has no relation to the sum type that we pattern match over. Therefore, when we type check the pattern match we need some type to check these I-terms against (like T-ILAM) rather than generating a type (like T-APP).

Figure 4.4 gives the rules for synthesizing sums in $\lambda^{\rightarrow}_{syn}$. Synthesizing injections proceeds similarly to synthesizing constants in $\lambda^{\rightarrow}_{syn}$. By assuming that the examples are well-typed, we know that they are some collection of inl or inr values. We are able to synthesize a inl value (IREFINE-SUM-INL) or inr value (IREFINE-SUM-INR) only when all of the values agree on their head constructor. In other words, it is safe to synthesize a constructor whenever the examples show that we can safely peel away the top-most constructor.

Synthesizing pattern matches (IREFINE-MATCH) proves to be a more complicated affair. We proceed in three steps:

1. Guess a value of sum type to pattern match against.

[2]We also elide the type annotations from the external language to unify the syntax of inl and inr although now their typing rules (T-INL and T-INR) must now guess the appropriate sum type. In an actual implementation, we would be type checking these terms in a bidirectional style where the sum type is given as input, so this is not a problem.

$\boxed{\Gamma \vdash \tau \rhd X \rightsquigarrow I}$

$$\textsc{irefine-sum-inl}\quad \frac{\Gamma \vdash \tau_1 \rhd \overline{\sigma_i \mapsto \chi_i}^{i<n} \rightsquigarrow I}{\Gamma \vdash \tau_1 + \tau_2 \rhd \overline{\sigma_i \mapsto \mathsf{inl}\,\chi_i}^{i<n} \rightsquigarrow \mathsf{inl}\,I}$$

$$\textsc{irefine-sum-inr}\quad \frac{\Gamma \vdash \tau_2 \rhd \overline{\sigma_i \mapsto \chi_i}^{i<n} \rightsquigarrow I}{\Gamma \vdash \tau_1 + \tau_2 \rhd \overline{\sigma_i \mapsto \mathsf{inr}\,\chi_i}^{i<n} \rightsquigarrow \mathsf{inr}\,I}$$

$$\textsc{irefine-match}\quad \frac{\begin{array}{c}\Gamma \vdash \tau_1 + \tau_2 \rightsquigarrow E \\ \mathbf{distribute}(E, X) = (X_l, X_r) \\ x_1{:}\tau_1, \Gamma \vdash \tau \rhd X_l \rightsquigarrow I_1 \qquad x_2{:}\tau_2, \Gamma \vdash \tau \rhd X_r \rightsquigarrow I_2\end{array}}{\Gamma \vdash \tau \rhd X \rightsquigarrow \mathsf{match}\ E\ \mathsf{with}\ \mathsf{inl}\,x_1 \to I_1 \mid \mathsf{inr}\,x_2 \to I_2}$$

$\boxed{v \simeq \chi}$

$$\textsc{eq-inl}\quad \frac{v \simeq \chi}{\mathsf{inl}\,v \simeq \mathsf{inl}\,\chi} \qquad \textsc{eq-inr}\quad \frac{v \simeq \chi}{\mathsf{inr}\,v \simeq \mathsf{inr}\,\chi}$$

$\mathbf{distribute}(E, X) = (X_l, X_r)$
where
$X_l = \{[v/x_1]\sigma \mapsto \chi \mid \sigma \mapsto \chi \in X \wedge \sigma(E) \longrightarrow^* \mathsf{inl}\,v\}$
$X_r = \{[v/x_2]\sigma \mapsto \chi \mid \sigma \mapsto \chi \in X \wedge \sigma(E) \longrightarrow^* \mathsf{inr}\,v\}$

Figure 4.4: $\lambda^{\rightarrow}_{syn}$ sums: synthesis rules

2. Distribute the example worlds among the branches of the pattern match.

3. Recursively synthesize the branches of the pattern match.

The **distribute** function accomplishes the second step. To distribute the examples, we evaluate the scrutinee expression discovered in step (1) under each of the example worlds. Because the examples are well-typed, we know that each such evaluation results in either an inl or inr value. We then distribute all of the example worlds that produce an inl value to the inl branch and all the example worlds that produce a inr value to the inr branch. We update each of these example worlds with a binding for the value contained in the constructor, but otherwise leave the goal example value untouched.

To make this process concrete, consider the following set of example worlds:

$$X = [\mathsf{inl}\, c_1 / x] \mapsto c_2, [\mathsf{inr}\, c_3 / x] \mapsto c_4, [\mathsf{inl}\, c_5 / x] \mapsto c_6.$$

If we guess the E-term x to pattern match against, then the examples are distributed as follows

$$\begin{aligned} X_1 &= [c_1 / x_1][\mathsf{inl}\, c_1 / x] \mapsto c_2, [c_5 / x_1][\mathsf{inl}\, c_5 / x] \mapsto c_6 \\ X_2 &= [c_3 / x_2][\mathsf{inl}\, c_3 / x] \mapsto c_4. \end{aligned}$$

We synthesize the inl branch expression using X_1 and the inr branch expression using X_2. Again, note that the example goal value has not changed during this process.

Finally, to prove soundness and completeness in the presence of sums, we require the following lemmas:

Lemma 4.3.1 (Type Preservation of distribute). *If* $\Gamma \vdash X : \tau$ *and* $\Gamma \vdash E \Leftarrow \tau_1 + \tau_2$ *then* $x_1{:}\tau_1, \Gamma \vdash X_l : \tau$ *and* $x_2{:}\tau_2, \Gamma \vdash X_r : \tau$ *where* $\textbf{distribute}(E, X) = (X_l, X_r)$.

Proof. By T-EXW-CONS, we know that each example world $\sigma \mapsto \chi \in X$ has type τ. By the definition of DISTRIBUTE, we know that each such example world is sent to either X_l or X_r with an additional binding, so it suffices to show that this additional binding is well-typed. Consider a single example world $\sigma \mapsto \chi \in X$. Unfolding the definition of **distribute** we see that we obtain the binding for this example world from:

$$\sigma(E) \to^* \mathsf{inx}\, v$$

where $\mathsf{inx}\, v$ is either $\mathsf{inl}\, v$ or $\mathsf{inr}\, v$. We know this evaluation is possible because of type safety, canonical forms, and strong normalization.

Consider the case where E evaluates to $\mathsf{inl}\, v$; the inr case proceeds identically. We know that v must have type τ_1 because E and $\mathsf{inl}\, v$, by preservation, have type $\tau_1 + \tau_2$. Therefore, by T-ENV-CONS, we know the binding is well-typed. □

Lemma 4.3.2 (Satisfaction Soundness of distribute). *Let I be the expression*

$$\begin{array}{l}\mathsf{match}\ E\ \mathsf{with}\\ \mid\ \mathsf{inl}\, x_1 \to I_1\\ \mid\ \mathsf{inr}\, x_2 \to I_2.\end{array}$$

If $I_1 \vDash X_l$ and $I_2 \vDash X_r$ and $\Gamma \vdash E \Leftarrow \tau_1 + \tau_2$ then $I \vDash X$ where **distribute**$(E, X) = (X_l, X_r)$.

Proof. Consider a single example world $\sigma \mapsto \chi \in X$. Because E is well-typed, we know by type safety, strong normalization, and canonical forms that $\sigma(E)$ reduces to either $\mathsf{inl}\, v$ or $\mathsf{inr}\, v$. Consider the inl case; the inr case proceeds identically. If $\sigma(E) \longrightarrow^* \mathsf{inl}\, v$, then by EVAL-MATCH-INL we must show that $v' \simeq \chi$ where $\sigma(I) \longrightarrow [v/x_1]\sigma(I_1) \longrightarrow^* v'$. However, by unfolding the definition of **distribute**, we know that $[v/x_1]\sigma \mapsto \chi \in X_l$. Because $I_1 \simeq X_l$, we can conclude that $v' \simeq \chi$. □

Lemma 4.3.3 (Satisfaction Preservation of distribute). *Let I be the expression*

$$\begin{array}{l}\mathsf{match}\ E\ \mathsf{with}\\ \mid\ \mathsf{inl}\, x_1 \to I_1\\ \mid\ \mathsf{inr}\, x_2 \to I_2.\end{array}$$

If $I \vDash X$ and $\Gamma \vdash E \Leftarrow \tau_1 + \tau_2$ then $I_1 \vDash X_l$ and $I_2 \vDash X_r$ where **distribute**$(E, X) = (X_l, X_r)$.

Proof. Consider a single example world $\sigma \mapsto \chi \in X$. Because E is well-typed, we know by type safety, strong normalization, and canonical forms that $\sigma(E)$ reduces to either $\mathsf{inl}\, v$ or $\mathsf{inr}\, v$. Consider the inl case; the inr case proceeds identically. If $\sigma(E) \longrightarrow^* \mathsf{inl}\, v$, then by EVAL-MATCH-INL and the fact that $I \vDash X$, $\sigma(I) \longrightarrow [v/x_1]\sigma(I_1) \longrightarrow *v'$ and $v' \simeq \chi$. Note that every such $[v/x_1]\sigma \mapsto \chi \in X_l$ by the definition of **distribute**. Therefore, we can conclude that $I_1 \vDash X_l$. □

4.3.1 Example: Boolean Operators

With sums, we can now synthesize much more realistic programs. For example, let's encode booleans in a more standard style using sums with

$$\begin{aligned}\mathsf{bool} &\stackrel{\text{def}}{=} T + T\\ \mathsf{true} &\stackrel{\text{def}}{=} \mathsf{inl}\, c\\ \mathsf{false} &\stackrel{\text{def}}{=} \mathsf{inr}\, c\end{aligned}$$

where we equipped T with a single constant c. Now consider the following set of example values for a binary operation:

$$\begin{aligned}\mathsf{true} &\Rightarrow \mathsf{true} \Rightarrow \mathsf{true}\\ \mathsf{false} &\Rightarrow \mathsf{true} \Rightarrow \mathsf{false}\\ \mathsf{true} &\Rightarrow \mathsf{false} \Rightarrow \mathsf{false}\\ \mathsf{false} &\Rightarrow \mathsf{false} \Rightarrow \mathsf{false}\end{aligned}$$

Let's derive the and function implied by these examples. After two applications of the IREFINE-ARR rule to remove the arrows, we arrive at the following synthesis state:

$$x{:}\mathsf{bool}, y{:}\mathsf{bool} \vdash \mathsf{bool} \rhd X' \rightsquigarrow \lambda x{:}\mathsf{bool}.\, \lambda y{:}\mathsf{bool}.\, \blacksquare$$

where

$$\begin{aligned}X' = {} & [\mathsf{true}/x][\mathsf{true}/y] \mapsto \mathsf{true}\\ , & [\mathsf{false}/x][\mathsf{true}/y] \mapsto \mathsf{false}\\ , & [\mathsf{true}/x][\mathsf{false}/y] \mapsto \mathsf{false}\\ , & [\mathsf{false}/x][\mathsf{false}/y] \mapsto \mathsf{false}.\end{aligned}$$

We now apply IREFINE-MATCH, guessing x to pattern match on which results in the following distribution of examples:

$$\begin{aligned}X_l = {} & [c/x_1][\mathsf{true}/x][\mathsf{true}/y] \mapsto \mathsf{true}\\ , & [c/x_1][\mathsf{true}/x][\mathsf{false}/y] \mapsto \mathsf{false}\\ X_r = {} & [c/x_2][\mathsf{false}/x][\mathsf{true}/y] \mapsto \mathsf{false}\\ , & [c/x_2][\mathsf{false}/x][\mathsf{false}/y] \mapsto \mathsf{false}\end{aligned}$$

and the hole filled in with the following program fragment:

$$\mathsf{match}\ x\ \mathsf{with}\ \mathsf{inl}\, x_1 \to \blacksquare \ \mid\ \mathsf{inr}\, x_2 \to \blacksquare$$

Synthesizing the inr branch is straightforward. We synthesize false for this branch because all the examples agree that the synthesized program should be false, *i.e.*, all the example values are false. Note that this proceeds in two derivation steps. In the first step, we apply IREFINE-SUM-INR because all of the goal example values are of the form of $\mathsf{inr}\, \chi$. In the second step, we apply IREFINE-BASE because all of the goal example values left are c.

Synthesizing the inl branch is slightly trickier. We note that we cannot EGUESS an E-term that satisfies X_1. Furthermore, we cannot apply IREFINE-SUM-INL because the head constructors of the examples do not match. Therefore, we must apply IREFINE-MATCH one more time, pattern matching on y, to distribute the examples

further

$$X_{ll} = [c/x_1'][c/x_1][\mathsf{true}/x][\mathsf{true}/y] \mapsto \mathsf{true}$$
$$X_{lr} = [c/x_1][\mathsf{true}/x][\mathsf{false}/y] \mapsto \mathsf{false}$$

where our program now looks like

$$\begin{array}{l}
\lambda x{:}\mathsf{bool}.\, \lambda y{:}\mathsf{bool}.\, \mathsf{match}\ x\ \mathsf{with} \\
\quad \mid\ \mathsf{inl}\, x_1 \rightarrow \\
\quad\ (\mathsf{match}\ y\ \mathsf{with} \\
\quad\quad \mid\ \mathsf{inl}\, x_1' \rightarrow \blacksquare \\
\quad\quad \mid\ \mathsf{inr}\, x_2' \rightarrow \blacksquare) \\
\quad \mid\ \mathsf{inr}\, x_2 \rightarrow \mathsf{false}.
\end{array}$$

In each of these branches, we can either guess the satisfying E-terms x and y with IREFINE-EGUESS, or we can apply IREFINE-SUM-INL and IREFINE-SUM-INR to synthesize true and false directly because there is only a single example in each branch. In either case, the final result is the usual implementation of and that we expect:

$$\begin{array}{l}
\lambda x{:}\mathsf{bool}.\, \lambda y{:}\mathsf{bool}.\, \mathsf{match}\ x\ \mathsf{with} \\
\quad \mid\ \mathsf{inl}\, x_1 \rightarrow \\
\quad\ (\mathsf{match}\ y\ \mathsf{with} \\
\quad\quad \mid\ \mathsf{inl}\, x_1' \rightarrow \mathsf{true} \\
\quad\quad \mid\ \mathsf{inr}\, x_2' \rightarrow \mathsf{false}) \\
\quad \mid\ \mathsf{inr}\, x_2 \rightarrow \mathsf{false}.
\end{array}$$

We can synthesize the other $2^3 - 1 = 7$ possible boolean operators using appropriate example sets with similar derivations in $\lambda_{syn}^{\rightarrow}$.

4.3.2 Efficiency of Sums

In Section 4.3.1, note the IREFINE-MATCH rule was highly non-deterministic in two dimensions:

1. We guessed an arbitrary E-term to pattern match against.
2. We could pattern match at any time because IREFINE-MATCH applies at any type τ.

Both dimensions introduce extreme inefficiencies into an implementation of synthesizer!

Building on the first point, not only can we pattern match against any E term, we did not restrict ourselves from pattern matching on the same term twice! For

example, while the following partial derivation

$$
\begin{array}{l}
\mathsf{match}\ x\ \mathsf{with} \\
\mid\ \mathsf{inl}\, x_1 \rightarrow \\
\quad \mid\ \mathsf{match}\ x\ \mathsf{with} \\
\qquad \mid\ \mathsf{inl}\, x_1' \rightarrow \mathsf{true} \\
\quad \ldots
\end{array}
$$

is perfectly sound, it results in unnecessary work because the inner pattern match duplicates the efforts of the outer pattern match. Such redundant pattern matches are obvious, but with a richer language to draw from, the problem becomes more subtle. For example, consider synthesizing programs over lists defined in the standard recursive style along with an append function that appends lists. The following pattern matches are semantically redundant:

$$
\begin{array}{r}
\mathsf{match}\ l\ \mathsf{with}\ \ldots \\
\mathsf{match}\ \mathsf{append}\ l\ []\ \mathsf{with}\ \ldots \\
\mathsf{match}\ \mathsf{append}\ []\ l\ \mathsf{with}\ \ldots
\end{array}
$$

but are not obviously redundant unless you crack open the definition of append.

Building on the second point, because we can invoke IREFINE-MATCH at any point in an I-refinement, we can now get into situations such as this:

$$
\begin{array}{l}
\mathsf{match}\ x\ \mathsf{with} \\
\quad \ldots \\
\quad \mathsf{match}\ y\ \mathsf{with} \\
\qquad \ldots \\
\\
\mathsf{match}\ y\ \mathsf{with} \\
\quad \ldots \\
\quad \mathsf{match}\ x\ \mathsf{with} \\
\qquad \ldots
\end{array}
$$

Here, we fall in the trap of synthesizing pattern matches over x and y, but we may do them in any order. And furthermore, those pattern matches may appear far apart in any branch of the program.

Historically, conditionals such as if statements and pattern matches have proven to be the most difficult to reason about during synthesis [Albarghouthi et al., 2013]. They not only represent points of non-determinism, they also greatly expand the branching factor of the program. Furthermore many conditionals are syntactically distinct but semantically equivalent, making search difficult to optimize. In Chapter 7, we go through significant lengths to minimize these points of inefficiencies.

4.4 Let Binding

So far we have added basic types and language features to $\lambda^{\rightarrow}_{syn}$ with good results. These features have admitted natural forms of examples and refinement rules and have not disrupted soundness or completeness of $\lambda^{\rightarrow}_{syn}$. However, the fact that a feature is simple does not necessarily mean that we can synthesize it easily. Let bindings are an excellent example of this fact.

Let bindings allow us to bind a value to a name, a function or some other type. They are useful for implementing helper functions that cannot be inlined into the main function's definition, *e.g.*because it is recursive, or simply shrinking the size of the program by avoiding code duplication. At first glance, we might introduce let bindings with the standard syntactic sugar:

$$\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \stackrel{\text{def}}{=} (\lambda x{:}\tau.\, e_2)\ e_1.$$

This is perfectly serviceable for the external language e of $\lambda^{\rightarrow}_{syn}$ which would allow us to use let-bindings in helper functions that we might feed to the synthesizer. However, this doesn't allow us to synthesize lets because the de-sugaring is not in normal form.

To get around this, we might introduce let as a standard syntactic form with the typing rule:

$$\textsc{t-Ilet}\ \frac{\Gamma \vdash I_1 \Leftarrow \tau_1 \qquad x{:}\tau_1, \Gamma \vdash I_2 \Leftarrow \tau_2}{\Gamma \vdash \mathsf{let}\ x{:}\tau_1 = I_1\ \mathsf{in}\ I_2 \Leftarrow \tau_2}$$

Transforming this into a synthesis rule, we obtain:

$$\textsc{irefine-let}\ \frac{\Gamma \vdash \tau_1 \rhd \cdot \rightsquigarrow I_1 \qquad X' = \ldots \quad x{:}\tau_1, \Gamma \vdash \tau_2 \rhd X' \rightsquigarrow I_2}{\Gamma \vdash \tau_2 \rhd X \rightsquigarrow \mathsf{let}\ x{:}\tau_1 = I_1\ \mathsf{in}\ I_2}$$

On top of the fact that let is not type-directed—IREFINE-LET applies at any type, similarly to IREFINE-MATCH—we must guess the "helper" type and term τ_1 and I_1 out of thin air!

Appealing to the Curry-Howard Isomorphism, synthesizing a let-binding is tantamount to guessing and deriving a lemma and then using that lemma in your proof. In the programming world, this is like guessing and deriving a helper function to use in the solution of a problem. IREFINE-LET precisely reflects our intuition about this process: coming up with a seemingly unrelated lemma or helper function is frequently as hard, if not harder, than solving the original problem itself! We leave this difficult problem of discovering and employing such let bindings to future work.

Chapter 5

Recursion

So far, we have considered adding a variety of basic types to $\lambda^{\to}_{syn}$. While these types allow us to synthesize programs that more closely match those found in actual functional programming languages, they have not significantly changed the expressiveness of our core language. Next we will consider adding recursion to $\lambda^{\to}_{syn}$ which greatly increases its expressiveness.

5.1 μ-types

One way to express recursion within a typed lambda calculus is with μ-types. Figure 5.1 shows how we can add μ-types to $\lambda^{\to}_{syn}$. The type $\mu\alpha.\,\tau$ binds a recursive occurrence of a type to the type variable α which appears in its definition τ. We introduce a μ-type with the fold term and eliminate a μ-type with the unfold term. For example, we may use μ-types, pair, sums, and Unit to encode a list type:

$$\mathsf{List} \stackrel{\text{def}}{=} \mu\alpha.\,\mathsf{Unit} + T * \alpha.$$

folds in a term explicitly mark points where the recursive type variable is used:

$$\mathsf{fold}\ \mathsf{inl}\ (c_1, \mathsf{fold}\ \mathsf{inl}\ (c_2, \mathsf{fold}\ \mathsf{inr}\ ())).$$

To use these recursive types, we explicitly unfold the folds where ever they appear, for example:

$$\begin{array}{l}(\mathsf{match}\ \mathsf{unfold}\ \mathsf{fold}\ \mathsf{inl}\ (c_1, \mathsf{fold}\ \mathsf{inr}\ ())\ \mathsf{with}\\ \quad \mathsf{inl}\ x_1 \to \mathsf{fst}\ x_1\\ \quad \mathsf{inr}\ x_2 \to c_2) \longrightarrow^* c_1.\end{array}$$

Synthesis rules, again, are derivable directly from the typing rules. To generate an unfold (EGUESS-UNFOLD), it is sufficient to guess a μ-type whose one-step unrolling is the goal type in question. When refining at μ-type (IGUESS-MU), we know that all our examples are fold values. Refining such a value is straightforward;

$$
\begin{array}{lcll}
\tau & ::= & \ldots \mid \alpha \mid \mu\alpha.\,\tau & \text{Types} \\
e & ::= & \ldots \mid \mathsf{fold}\, e \mid \mathsf{unfold}\, e & \text{Terms} \\
v & ::= & \ldots \mid \mathsf{fold}\, v & \text{Values} \\
\mathcal{E} & ::= & \ldots \mid \mathsf{fold}\, \mathcal{E} \mid \mathsf{unfold}\, \mathcal{E} & \text{Evaluation Contexts} \\
E & ::= & \ldots \mid \mathsf{unfold}\, E & \text{Elimination Terms} \\
I & ::= & \ldots \mid \mathsf{fold}\, I & \text{Introduction Terms} \\
\chi & ::= & \ldots \mid \mathsf{fold}\, \chi & \text{Example Values}
\end{array}
$$

$\boxed{\Gamma \vdash e : \tau}$

$$\textsc{t-fold}\ \frac{\Gamma \vdash e : [\mu\alpha.\,\tau/\alpha]\tau}{\Gamma \vdash \mathsf{fold}\, e : \mu\alpha.\,\tau} \qquad \textsc{t-unfold}\ \frac{\Gamma \vdash e : \mu\alpha.\,\tau}{\Gamma \vdash \mathsf{unfold}\, e : [\mu\alpha.\,\tau/\alpha]\tau}$$

$\boxed{\Gamma \vdash E \Rightarrow \tau}$

$$\textsc{t-unfold}\ \frac{\Gamma \vdash E \Rightarrow \mu\alpha.\,\tau}{\Gamma \vdash \mathsf{unfold}\, E \Rightarrow [\mu\alpha.\,\tau/\alpha]\tau}$$

$\boxed{\Gamma \vdash I \Leftarrow \tau}$

$$\textsc{t-Ifold}\ \frac{\Gamma \vdash I \Leftarrow [\mu\alpha.\,\tau]\tau}{\Gamma \vdash \mathsf{fold}\, I \Leftarrow \mu\alpha.\,\tau}$$

$\boxed{e \longrightarrow e'}$

$$\textsc{eval-unfold-fold}\ \mathsf{unfold}\, \mathsf{fold}\, v \longrightarrow v$$

$\boxed{\Gamma \vdash \tau \rightsquigarrow E}$

$$\textsc{eguess-unfold}\ \frac{\Gamma \vdash \mu\alpha.\,\tau \Rightarrow E}{\Gamma \vdash [\mu\alpha.\,\tau/\alpha]\tau \Rightarrow \mathsf{unfold}\, E}$$

$\boxed{\Gamma \vdash \tau \rhd X \rightsquigarrow I}$

$$\textsc{irefine-mu}\ \frac{X = \overline{\sigma_i \mapsto \mathsf{fold}\, \chi_i}^{i<n} \qquad \Gamma \vdash [\mu\alpha.\,\tau/\alpha]\tau \rhd \mathbf{unfold}(X') \rightsquigarrow I}{\Gamma \vdash \mu\alpha.\,\tau \rhd X \rightsquigarrow \mathsf{fold}\, I}$$

$\boxed{\Gamma \vdash \chi : \tau}$

$$\textsc{t-ex-fold}\ \frac{\Gamma \vdash \chi : [\mu\alpha.\,\tau/\alpha]\tau}{\Gamma \vdash \mathsf{fold}\, \chi : \mu\alpha.\,\tau}$$

$\boxed{v \simeq \chi}$

$$\textsc{eq-pair}\ \frac{v \simeq \chi}{\mathsf{fold}\, v \simeq \mathsf{fold}\, \chi}$$

$$\mathbf{unfold}(\overline{\sigma_i \mapsto \mathsf{fold}\, \chi_i}^{i<n}) = \overline{\sigma_i \mapsto \chi_i}^{i<n}$$

Figure 5.1: $\lambda^{\rightarrow}_{syn}$ μ types

because folds have no computational content, we simply shed the top-level fold constructors of the examples, leaving behind example values of an appropriate type that we can use to synthesize the unfolded program.

Up until now, I-refinement has always decomposed down a goal type down into a base type. At first glance, μ-types seem to pose a problem for type-directed example refinement because a μ-type can be unfolded infinitely, for example, our List type:

$$\begin{aligned} & \mu\alpha.\,\mathsf{Unit} + T * \alpha \\ \equiv\ & \mathsf{Unit} + (T * \mu\alpha.\,\mathsf{Unit} + T * \alpha) \\ \equiv\ & \mathsf{Unit} + (T * \mathsf{Unit} + (T * \mu\alpha.\,\mathsf{Unit} + T * \alpha)) \\ \equiv\ & \ldots \end{aligned}$$

However, example values save us because while a μ-type represents an infinite family of types, example values are necessarily finite structures. This simple example value of type List:

$$\mathsf{fold}\ \mathsf{inl}\ (c, \mathsf{fold}\ \mathsf{inr}\ ())$$

Can only be unfolded twice, corresponding to the two folds in the value. Thus application of IREFINE-MU stop once all the folds have been peeled away from the example values.

5.1.1 Non-determinism

The rules in Figure 5.1 seem perfectly sensible. Indeed, they are sound as we can show with the appropriate lemma.

Lemma 5.1.1 (Example-Type Preservation of unfold). *If* $\Gamma \vdash X : \mu\alpha.\,\tau$ *then* $\Gamma \vdash \mathbf{unfold}(X) : [\mu\alpha.\,\tau/\alpha]\tau$.

Proof. Immediate from our premise. T-EXW-CONS says that each σ_i is well-typed and T-EX-FOLD says that each example fold is well typed along with their components at type $[\mu\alpha.\,\tau/\alpha]\tau$. □

It turns out that the necessary completeness lemma also holds rather trivially:

Lemma 5.1.2 (Satisfaction Preservation of unfold). *If* $\mathsf{fold}\ I \vDash X$ *then* $I \vDash \mathbf{unfold}(X)$.

Proof. Consider a single example world $\sigma \mapsto \mathsf{fold}\ \chi$. By SATISFIES and EQ-FOLD, we know that $\sigma(I) \simeq \chi$. We know from the definition of **unfold** that $\forall \sigma \mapsto \mathsf{fold}\ \chi.\, \sigma \mapsto \chi \in \mathbf{unfold}(X)$ which is sufficient to conclude that $I \vDash \mathbf{unfold}(X)$. □

The problem is that we know that the addition of μ-types into the $\lambda^{\to}$ introduces non-termination [Pierce, 2002]! In particular, let ω be diverging term. Then the

expression $\lambda x : \tau.\, \omega$ cannot be synthesized in $\lambda^{\rightarrow}_{syn}$ extended with μ-types. Because IREFINE-GUESS relies on evaluation, we will never be able to use it to synthesize ω (which is an E-term). Thus in the presence of recursion, we lose completeness.

However, recursion poses even more of a problem than this. It is is unlikely for us to encounter ω if we enumerate programs in order of size as suggested in Chapter 2 because its encoding in $\lambda^{\rightarrow}_{syn}$ is roughly 30 AST nodes which is too large to generate in a reasonable amount of time. However, such non-terminating expressions in a standard functional programming language are much smaller by comparison. For example imagine that we are synthesizing the body of a function in an ML-like language:

$$\mathsf{let\ rec}\ f\ (x{:}\mathsf{nat}) : \mathsf{nat} = \blacksquare.$$

We may try to E-guess the expression $f\,x$ for the body of f which produces an infinite loop. This expression, in contrast to ω, is a mere 3 AST nodes which makes it very likely that we would encounter this term during enumeration. Furthermore, because this looping term is so small, we'll enumerate many other equivalent terms that contain it, *e.g.*, $f\,(f\,x)$ or $f\,(f\,(f\,x))$. We could alter our evaluation strategy to include a timeout or limit on number of evaluation steps, but with so many non-terminating terms, this approach would not scale appropriately.

5.2 A Functional Synthesis Programming Language

Because we can easily enumerate infinite loops that ruin our generate-and-test E-guess strategy, we need some way of restricting recursion so that we can regain strong normalization. Rather than try to solve this problem within $\lambda^{\rightarrow}_{syn}$ where we do not have many hooks for limiting recursive definitions, we graduate to ML_{syn}—a core ML-like calculus for program synthesis—and adopt standard techniques to obtain strong normalization in the system. Figure 5.2 defines the syntax of ML_{syn} which is divided into an external language of expressions e and an internal language of introduction terms I and elimination terms E that constitute the β-normal forms of e.

ML_{syn} more closely approximates a real-world functional programming language like OCaml, Haskell, or F# [Leroy et al., 2014; Peyton Jones et al., 2003; Syme, 2013]. On top of the lambdas and application introduced in $\lambda^{\rightarrow}_{syn}$, ML_{syn} introduces two major features:

1. Algebraic data types and
2. Recursive function definitions.

Algebraic Data Types Rather than sums and products, ML_{syn} features algebraic data types defined by a signature of constructors Σ. Constructors C are defined to

$$
\begin{array}{lcl}
\tau & ::= & T \mid \tau_1 \to \tau_2 \\
p & ::= & C(x_1, \ldots, x_k) \\
\rho & ::= & \overline{v_i \Rightarrow \chi_i}^{i<m} \\
e & ::= & x \mid e_1\ e_2 \mid \rho \mid \mathsf{fix}\ f\ (x{:}\tau_1) : \tau_2 = e \mid C\ (e_1, \ldots, e_k) \\
 & \mid & \mathsf{match}\ e\ \mathsf{with}\ \overline{p_i \to e_i}^{i<m} \mid \mathsf{NoMatch} \\
v & ::= & \rho \mid \mathsf{fix}\ f\ (x{:}\tau_1) : \tau_2 = e \mid C\ (v_1, \ldots, v_k) \mid \mathsf{NoMatch} \\
\mathcal{E} & ::= & \square \mid \mathcal{E}\ e \mid v\ \mathcal{E} \mid C\ (v_1, \ldots, \mathcal{E}, \ldots, e_k) \\
 & \mid & \mathsf{match}\ \mathcal{E}\ \mathsf{with}\ \overline{p_i \to e_i}^{i<m} \\
b & ::= & \cdot \mid \mathsf{rec} \mid \mathsf{arg}\, f \mid \mathsf{dec}\, f \\
\Gamma & ::= & \cdot \mid x{:}\tau\{b\}, \Gamma \\
\Sigma & ::= & \cdot \mid C{:}\tau_1 * \ldots * \tau_k \to T, \Sigma \\
 & & \\
E & ::= & x \mid E\ I \\
I & ::= & E \mid \mathsf{fix}\ f\ (x{:}\tau_1) : \tau_2 = I \\
 & \mid & C(I_1, \ldots, I_k) \mid \mathsf{match}\ E\ \mathsf{with}\ \overline{p_i \to I_i}^{i<m} \\
\sigma & ::= & \cdot \mid [v/x]\sigma \\
\chi & ::= & \rho \mid C(\chi_1, \ldots, \chi_k) \\
X & ::= & \cdot \mid \sigma \mapsto \chi, X
\end{array}
$$

Figure 5.2: ML_{syn} syntax

take a tuple of arguments of type $\tau_1 * \ldots * \tau_k$ and produce a value of a particular data type T. This tuple may be empty in which case we define a nullary constructor, *i.e.*, a constructor that takes no arguments and is directly a value of type T. We constrain patterns to be of the form $C(x_1, \ldots, x_k)$, disallowing nested patterns in order to simplify our presentation without loss of expressiveness.

Note that this particular formulation of constructors implies that a constructor $C(e_1, \ldots, e_k)$ must be fully saturated, that is, it is always provided all its arguments. This choice mirrors OCaml and F# which also require that all constructor values are fully saturated, but stands in opposition to Haskell which treats a constructor as a function $\tau_1 \to \ldots \to \tau_k \to T$ that may be partially applied. While this design choice has implications for the implementation of data types in these languages, it does not affect synthesis in any significant way as we have demonstrated that our synthesis approach handles both tuples and partially applied functions without problems.

Recursive Functions In addition to algebraic data types, ML_{syn} also features recursive function values: $\mathsf{fix}\ f\ (x{:}\tau_1) : \tau_2 = e$. By default ML_{syn} synthesizes recursive functions where e may mention the function f. If f is not in the free variables of e, then we use the usual lambda notion $\lambda x{:}\tau_1.\, e$ as syntactic sugar.

Recall from Section 2.3.4 that we synthesized a (non-recursive) function by

splitting input/output examples (realized as partial function example values). This way, we are able to provide a value binding for each argument that we record in each example world's environment σ. Because the functions we synthesize will be recursive, we expect to add a binding for the function f itself in addition to its argument. But then, what value binding can we provide for f, the function we are currently synthesizing?

As we discuss in detail in Section 5.2.2, we use the partial function that specifies f as its own value. To do this, we promote partial function values from mere example values to (external language) values. In $\lambda^{\rightarrow}_{syn}$, example values χ were a distinct grammar production from v. The two productions differed in their representation of values at arrow types: example values provide partial function terms $\overline{\chi_i \Rightarrow v_i}^{i<m}$ and proper values provide lambdas $\lambda x{:}t.\, e$. In ML_{syn}, the grammar of example values χ is now a proper subset of the values. We continue to maintain that χ does not include lambdas so that functions cannot appear in the goal position of input/output pairs. This design decision has important ramifications for the metatheory of ML_{syn} that we discuss in Chapter 6.

5.2.1 Static and Dynamic Semantics of ML_{syn}

Figure 5.3 and Figure 5.4 gives the rules for type checking the external and internal languages of ML_{syn}, respectively. With some minor differences, the rules mirror the type checking rules of $\lambda^{\rightarrow}_{syn}$:

1. To admit recursive definitions, T-FIX introduces a binder for the function value f in addition to its argument x.

2. Expanding on the sum rule which binds a single variable, a constructor pattern binds an arbitrary number of variables. T-MATCH delegates this work to the **binders** meta-function.

In addition to these standard rules, we enforce three additional properties to ensure totality of the system:

Structural recursion: To ensure that a chain of recursive function calls always terminates, we require that each recursive call is made to a structurally smaller argument. This way, we know that the recursion bottoms out once we have completely decomposed a value. To track this information, we introduce *binding specifications* b to the bindings in Γ and update them as we type check a program. There are four possible binding specifications:

- $\cdot$: no specification,
- rec: denotes a recursive function,
- arg f: denotes an argument to a function f, and

$\boxed{\Sigma;\Gamma \vdash e : \tau}$

$$\textsc{T-VAR}\ \frac{x{:}\tau \in \Gamma}{\Sigma;\Gamma \vdash x : \tau} \qquad \textsc{T-APP}\ \frac{\Sigma;\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Sigma;\Gamma \vdash e_2 : \tau_1 \qquad \mathbf{struct}(\Gamma, e_1, e_2)}{\Sigma;\Gamma \vdash e_1\ e_2 : \tau_2}$$

$$\textsc{T-CTOR}\ \frac{C : \tau_1, \ldots, \tau_k \to T \in \Sigma \qquad \overline{\Sigma;\Gamma \vdash e_i : \tau_i}^{i<k}}{\Sigma;\Gamma \vdash C\ (e_1, \ldots, e_k) : T}$$

$$\textsc{T-PF}\ \frac{\overline{\Sigma;\Gamma \vdash v_i : \tau_1}^{i<m} \qquad \overline{\Sigma;\Gamma \vdash \chi_i : \tau_2}^{i<m}}{\Sigma;\Gamma \vdash \overline{v_i \Rightarrow \chi_i}^{i<m} : \tau_1 \to \tau_2} \qquad \textsc{T-FIX}\ \frac{\Sigma; f{:}\tau_1 \to \tau_2\{\mathsf{rec}\}, x{:}\tau_1\{\mathsf{arg}\ f\}, \Gamma \vdash e : \tau_2}{\Sigma;\Gamma \vdash \mathsf{fix}\ f\ (x{:}\tau_1) : \tau_2 = e : \tau_1 \to \tau_2}$$

$$\textsc{T-MATCH}\ \frac{\Sigma;\Gamma \vdash e : T \qquad \mathbf{complete}(\Sigma, \overline{p_i}^{i<m}, T) \qquad \overline{\mathbf{binders}(\Gamma, e, p_i) = \Gamma_i}^{i<m} \qquad \overline{\Sigma;\Gamma_i,\Gamma \vdash e_i : \tau}^{i<m}}{\Sigma;\Gamma \vdash \mathsf{match}\ e\ \mathsf{with}\ \overline{p_i \to e_i}^{i<m} : \tau} \qquad \textsc{T-NOMATCH}\ \frac{}{\Sigma;\Gamma \vdash \mathsf{NoMatch} : \tau}$$

$\boxed{\mathbf{struct}(e_1, e_2)}$

$$\textsc{STRUCT-VAR-REC}\ \frac{f{:}\tau_1 \to \tau_2\{\mathsf{rec}\} \in \Gamma \qquad x{:}\tau_1\{\mathsf{dec}\ f\} \in \Gamma}{\mathbf{struct}(\Gamma, f, x)}$$

$$\textsc{STRUCT-VAR-NOT-REC}\ \frac{f{:}\tau_1 \to \tau_2\{b\} \in \Gamma \qquad b \neq \mathsf{rec}}{\mathbf{struct}(\Gamma, f, e)} \qquad \textsc{STRUCT-NOT-VAR}\ \frac{e_1 \neq f}{\mathbf{struct}(\Gamma, e_1, e_2)}$$

$\boxed{\mathbf{complete}(\Sigma, \overline{p_i}^{i<m}, T)}$

$$\textsc{COMPLETE}\ \frac{C \in \tau_1 * \ldots * \tau_k \to T \in \Sigma \leftrightarrow C(x_1, \ldots, x_k) \in \overline{p_i}^{i<m}}{\mathbf{complete}(\Sigma, \overline{p_i}^{i<m})}$$

$\boxed{\mathbf{pos}(T, \tau)}$

$$\textsc{POS-BASE}\ \frac{}{\mathbf{pos}(T, T')} \qquad \textsc{POS-ARR}\ \frac{\tau_1 \neq T \qquad \mathbf{pos}(T, \tau_1) \qquad \mathbf{pos}(T, \tau_2)}{\mathbf{pos}(T, \tau_1 \to \tau_2)}$$

$\boxed{\vdash \Sigma}$

$$\textsc{SIG-EMPTY}\ \frac{}{\vdash \cdot} \qquad \textsc{SIG-CONS}\ \frac{\mathbf{pos}(T, \tau_1) \ldots \mathbf{pos}(T, \tau_k) \qquad \vdash \Sigma}{\vdash C{:}\tau_1 * \ldots * \tau_k, \Sigma}$$

$\mathbf{binders}(\Gamma, e, C(x_1, \ldots, x_k)) = x_1{:}\tau_1\{b_1\}, \ldots, x_k{:}\tau_k\{b_k\}$
where

$$\forall i \in 1, \ldots, k.\ b_i = \begin{cases} \mathsf{dec}\ f & e = x, x{:}\tau\{b\} \in \Gamma, b = \mathsf{arg}\ f \vee b = \mathsf{rec} f, \tau_i = \tau \\ \cdot & \text{otherwise} \end{cases}$$

Figure 5.3: ML_{syn} external language type checking

$\boxed{\Sigma;\Gamma \vdash E \Rightarrow \tau}$

$$\textsc{t-Evar}\ \frac{x{:}\tau \in \Gamma}{\Sigma;\Gamma \vdash x \Leftarrow \tau} \qquad \textsc{t-Eapp}\ \frac{\Sigma;\Gamma \vdash E \Rightarrow \tau_1 \to \tau_2 \qquad \Sigma;\Gamma \vdash I \Leftarrow \tau_1 \qquad \mathbf{struct}(\Gamma, E, I)}{\Sigma;\Gamma \vdash E\ I : \tau_2}$$

$\boxed{\Sigma;\Gamma \vdash I \Leftarrow \tau}$

$$\textsc{t-Ielim}\ \frac{\Sigma;\Gamma \vdash E \Rightarrow \tau}{\Sigma;\Gamma \vdash I \Leftarrow \tau} \qquad \textsc{t-Ictor}\ \frac{C : \tau_1, \ldots, \tau_k \to T \in \Sigma \qquad \overline{\Sigma;\Gamma \vdash I_i \Leftarrow \tau_i}^{i<k}}{\Sigma;\Gamma \vdash C\ (I_1, \ldots, I_k) \Leftarrow T}$$

$$\textsc{t-fix}\ \frac{\Sigma; f{:}\tau_1 \to \tau_2\{\mathsf{rec}\}, x{:}\tau_1\{\mathsf{arg}\ f\}, \Gamma \vdash I \Leftarrow \tau_2}{\Sigma;\Gamma \vdash \mathsf{fix}\ f\ (x{:}\tau_1) : \tau_2 = I \Leftarrow \tau_1 \to \tau_2}$$

$$\textsc{t-match}\ \frac{\Sigma;\Gamma \vdash E \Rightarrow T \qquad \mathbf{complete}(\Sigma, \overline{p_i}^{i<m}, T) \qquad \overline{\mathbf{binders}(\Gamma, E, p_i) = \Gamma_i}^{i<m} \qquad \overline{\Sigma;\Gamma_i, \Gamma \vdash I_i \Leftarrow \tau}^{i<m}}{\Sigma;\Gamma \vdash \mathsf{match}\ E\ \mathsf{with}\ \overline{p_i \to e_i}^{i<m} \Leftarrow \tau}$$

$\boxed{\mathbf{struct}(E, I)}$

$$\textsc{struct-var-rec}\ \frac{f{:}\tau_1 \to \tau_2\{\mathsf{rec}\} \in \Gamma \qquad x{:}\tau_1\{\mathsf{dec}\ f\} \in \Gamma}{\mathbf{struct}(\Gamma, f, x)}$$

$$\textsc{struct-var-not-rec}\ \frac{f{:}\tau_1 \to \tau_2\{b\} \in \Gamma \qquad b \neq \mathsf{rec}}{\mathbf{struct}(\Gamma, f, I)} \qquad \textsc{struct-not-var}\ \frac{E \neq f}{\mathbf{struct}(E, I)}$$

$\boxed{\mathbf{complete}(\Sigma, \overline{p_i}^{i<m}, T)}$

$$\textsc{complete}\ \frac{C \in \tau_1 * \ldots * \tau_k \to T \in \Sigma \leftrightarrow C(x_1, \ldots, x_k) \in \overline{p_i}^{i<m}}{\mathbf{complete}(\Sigma, \overline{p_i}^{i<m})}$$

$\mathbf{binders}(\Gamma, E, C(x_1, \ldots, x_k)) = x_1{:}\tau_1\{b_1\}, \ldots, x_k{:}\tau_k\{b_k\}$
where

$$\forall i \in 1, \ldots, k.\, b_i = \begin{cases} \mathsf{dec}\ f & E = x, x{:}\tau\{b\} \in \Gamma, b = \mathsf{arg}\ f \vee b = \mathsf{rec} f, \tau_i = \tau \\ \cdot & \text{otherwise} \end{cases}$$

Figure 5.4: ML_{syn} internal language type checking

- dec f: denotes a structurally decreasing variable to the function f.

When a binding has no specification, we avoid writing down the empty specification in favor of the standard binding syntax $x{:}\tau$.

The judgment $\mathbf{struct}(e_1, e_2)$ ensures that two expressions constitute a valid function application. Non-recursive functions applications are always valid (STRUCT-VAR-NOT-REC and STRUCT-NOT-VAR). If the function e_1 is a recursive function (currently being defined), *i.e.*, is a variable f annotated with rec, then the argument e_2 must be a variable annotated with dec f. We require that any recursive function application consists of a variable f annotated with rec and a variable x annotated with dec f. In particular, a recursive function application involving an argument that is not a variable is always invalid. This is strictly more constraining than Coq which performs some β-reductions to admit more programs [The Coq Development Team, 2012], but is much easier to formalize and implement in practice.

T-APP ensures that every application is structurally recursive (or non-recursive) with **struct**. When type checking a function with T-FIX, we record bindings for both the argument x and the function itself f. f is marked as a recursive function, rec, and the argument x is marked as an argument of f, arg f. The only way to obtain a structurally decreasing value of f is to pattern match on a variable that is either an argument of f, arg f, or is already structurally decreasing with respect to f, dec f. When computing the binders of a pattern match in T-MATCH, we appeal to the helper function $\mathbf{binders}(\Gamma, e, p)$ which creates binders for each of the pattern variables in p. When computing the binding specifications for each variable, we check to see if the scrutinee of the pattern match e is a variable that is annotated as arg f or dec f of some recursive function f. If it is, then any sub-component of that pattern is marked dec f if its type coincides with the type of e, *i.e.*, it is a recursive occurrence of a data type in a constructor.

Pattern completeness: We ensure that every pattern match over some data type T covers all the possible constructor values of T. Because we require patterns to be of the form $C(x_1, \ldots, x_k)$, this restriction amounts to having exactly one branch for each constructor of type T. This constraint is enforced by the judgment $\mathbf{complete}(\Sigma, \overline{p_i}^{i<m}, T)$ which ensures there is a one-to-one correspondence between a set of patterns and constructors at type T.

Positivity restriction: Finally, we enforce a positivity restriction on data types similar to Coq and Agda [Norell, 2007; The Coq Development Team, 2012] that ensures that a recursive occurrence of a data type does not occur to the left of an arrow in the type of a constructor. To see why this restriction is

necessary, consider the data type:

$$\mathsf{data}\ d = D\ \mathsf{of}\ d \to \tau$$

For some other data type τ. Now consider the function:

$$\begin{array}{l}\mathsf{let}\ f\ (x{:}d) : \tau = \\ \quad \mathsf{match}\ d\ \mathsf{with} \\ \quad \mid\ D\, g \to g\, d.\end{array}$$

Then the application $f\ (D\, f)$ is well-typed but goes into an infinite loop. The judgment $\mathbf{pos}(T, \tau)$ enforces this constraint for a particular argument type τ against a data type T. When checking that a signature is well-formed ($\vdash \Sigma$), we ensure that each argument of each constructor in Σ obeys the positivity restriction.

Figure 5.5 gives the evaluation rules and the compatibility judgment for the system. On top of the usual evaluations rules, we also provide evaluation rules for partial functions as they are now values. Partial functions values behave like a pattern match (EVAL-PF-GOOD): when applied to an argument v_j, the partial function $\overline{v_i \Rightarrow \chi_i}^{i<m}$ produces the output χ_j (where $j \in 1, \ldots, m$), taking advantage of the fact that example values are now a proper subset of the values.

However, if v_j are not among the $\overline{v_i}^{i<m}$ of the pattern match, evaluation produces a NoMatch exception value (EVAL-PF-BAD) which then becomes the overall result of the computation (EVAL-NOMATCH). The NoMatch value type checks at any type T (T-NOMATCH), and it only exists in the external language as it arises as the result of evaluating a partial function. Note that we've already restricted our match expressions to be complete (via the **complete** judgment), so NoMatch values can only arise due to pattern match failures when evaluating partial functions.

To determine which value to produce, we use the compatibility judgment ($\simeq$) that we used before to verify that an E-guessed term satisfies our examples. In $\lambda^{\rightarrow}_{syn}$ the compatibility judgment compared an example value χ and a value v. However, when evaluating partial functions, we will compare two values—the argument and the input of each alternative—so we must expand compatibility to compare two values. Now that the syntactic classes are the same on both sides, we use symmetry (EQ-SYM) to account for the new case when we compare a partial function on the left to a function on the right, *i.e.*, the flipped version of EQ-FIX-PF. We can now also compare two partial functions for compatibility. EQ-PF-PF says that two partial functions are compatible if and only if we can draw a one-to-one correspondence between their alternatives where compatible inputs produce compatible outputs. Finally, because we are comparing two values, we may end up comparing two functions for compatibility. In $\lambda^{\rightarrow}_{syn}$ such a comparison is possible because functions are non-recursive so we can resort to a strategy based off of $\beta\eta$-equivalence. However, in ML_{syn} this is not the case as our functions are recursive

$\boxed{e \to e'}$

EVAL-CTX
$$\frac{e \longrightarrow e'}{\mathcal{E}[e] \longrightarrow \mathcal{E}[e']}$$

EVAL-NOMATCH
$$\mathcal{E}[\mathsf{NoMatch}] \longrightarrow \mathsf{NoMatch}$$

EVAL-APP
$$(\lambda x{:}\tau.\, e)\ v \longrightarrow [v/x]e$$

EVAL-PF-GOOD
$$\frac{v \simeq v_j \qquad j \in 1, \ldots, m}{(\overline{v_i \Rightarrow \chi_i}^{\,i<m})\ v \to \chi_j}$$

EVAL-PF-BAD
$$\frac{\forall i \in 1, \ldots, m.\, v \not\simeq v_i}{(\overline{v_i \Rightarrow \chi_i}^{\,i<m})\ v \to \mathsf{NoMatch}}$$

EVAL-MATCH
$$\mathsf{match}\ C(v_1, \ldots, v_k)\ \mathsf{with}\ p_1 \to e_1 \mid \ldots \mid C(x_1, \ldots, x_k) \to e \mid \ldots \mid p_m \to e_m \longrightarrow [v_1/x_1]\ldots[v_k/x_k](e)$$

$\boxed{v \simeq v'}$

EQ-REFL
$$\frac{}{v \simeq v}$$

EQ-SYM
$$\frac{v' \simeq v}{v \simeq v'}$$

EQ-PF-PF
$$\frac{\begin{array}{c}\forall i \in 1, \ldots, m.\, \exists j \in 1, \ldots, n.\, v_i \simeq v_j \wedge \chi_i \simeq \chi_j \\ \forall j \in 1, \ldots, n.\, \exists i \in 1, \ldots, m.\, v_i \simeq v_j \wedge \chi_i \simeq \chi_j\end{array}}{\overline{v_i \Rightarrow \chi_i}^{\,i<m} \simeq \overline{v_j \Rightarrow \chi_j}^{\,j<n}}$$

EQ-FIX-PF
$$\frac{\forall i \in 1, \ldots, m.\, (\mathsf{fix}\ f\ (x{:}\tau_1) : \tau_2 = e)\ v_i \longrightarrow^* v \wedge v \simeq \chi_i}{\mathsf{fix}\ f\ (x{:}\tau_1) : \tau_2 = e \simeq \overline{v_i \Rightarrow \chi_i}^{\,i<m}}$$

EQ-CTOR
$$\frac{v_{11} \simeq v_{21} \ \ldots\ v_{1k} \simeq v_{2k}}{C(v_{11}, \ldots, v_{1k}) \simeq C(v_{21}, \ldots, v_{2k})}$$

Figure 5.5: ML_{syn} evaluation and compatibility rules

$\boxed{\Sigma;\Gamma \vdash \tau \rightsquigarrow E}$

$$\textsc{eguess-var}\ \frac{x{:}\tau \in \Gamma}{\Sigma;\Gamma \vdash \tau \rightsquigarrow x} \qquad \textsc{eguess-app}\ \frac{\Sigma;\Gamma \vdash \tau_1 \to \tau_2 \rightsquigarrow E \qquad \Sigma;\Gamma \vdash \tau_1 \rhd \cdot \rightsquigarrow I \qquad \mathbf{struct}(\Gamma, E, I)}{\Sigma;\Gamma \vdash \tau_2 \rightsquigarrow E\ I}$$

$\boxed{I \vDash X}$

$$\textsc{satisfies}\ \frac{\forall \sigma \mapsto \chi \in X.\, \sigma(I) \longrightarrow^* v \wedge v \simeq \chi}{I \vDash X}$$

$\boxed{\Sigma;\Gamma \vdash \tau \rhd X \rightsquigarrow I}$

$$\textsc{irefine-guess}\ \frac{\Sigma;\Gamma \vdash \tau \rightsquigarrow E \qquad E \vDash X}{\Sigma;\Gamma \vdash \tau \rhd X \rightsquigarrow E}$$

$$\textsc{irefine-base}\ \frac{C : \tau_1, \ldots, \tau_k \to T \in \Sigma \qquad X = \overline{\sigma_i \mapsto C(I_{i1}, \ldots, I_{ik})}^{i<n} \qquad \mathbf{proj}(X) = X_1, \ldots, X_k \qquad \overline{\Sigma;\Gamma \vdash \tau_j \rhd X_j \rightsquigarrow I_j}^{j<k}}{\Sigma;\Gamma \vdash T \rhd X \rightsquigarrow C\ (I_1, \ldots, I_k) \Leftarrow T}$$

$$\textsc{irefine-arr}\ \frac{\begin{array}{c} X = \sigma_1 \mapsto \rho_1, \ldots, \sigma_n \mapsto \rho_n \\ X' = \mathbf{apply}(f, x, \sigma_1 \mapsto \rho_1) +\!\!+ \ldots +\!\!+ \mathbf{apply}(f, x, \sigma_n \mapsto \rho_n) \\ \Sigma; f{:}\tau_1 \to \tau_2\{\mathsf{rec}\}, x{:}\tau_1\{\mathsf{arg}\, f\}, \Gamma \vdash I \rhd X' \rightsquigarrow \tau_2 \end{array}}{\Sigma;\Gamma \vdash \tau_1 \to \tau_2 \rhd X \rightsquigarrow \mathsf{fix}\ f\ (x{:}\tau_1) : \tau_2 = I}$$

$$\textsc{irefine-match}\ \frac{\Sigma;\Gamma \vdash T \rightsquigarrow E \qquad \mathbf{distribute}(\Sigma, T, X, E) = \overline{(p_i, X_i')}^{i<m} \qquad \overline{\mathbf{binders}(\Gamma, E, p_i) = \Gamma_i}^{i<m} \qquad \overline{\Sigma;\Gamma_i, \Gamma \vdash \tau \rhd X \rightsquigarrow I_i}^{i<m}}{\Sigma;\Gamma \vdash \tau \rhd X \rightsquigarrow \mathsf{match}\ E\ \mathsf{with}\ \overline{p_i \to e_i}^{i<m}}$$

Figure 5.6: ML_{syn} synthesis

by default; in general, determining compatibility in this case is undecidable (a point we discuss in Chapter 6). To remedy this, we add a reflexivity rule (EQ-REFL) that equates two syntactically identical terms. This is only an approximation, as we only consider a function equivalent to itself, but this works well in ML_{syn} because comparing syntactically distinct, yet equivalent functions only occurs at higher type and is, therefore, uncommon in practice.

5.2.2 Synthesis in ML_{syn}

Like $\lambda^{\to}_{syn}$, we simply adapt the internal language typing judgment to perform synthesis by making the type of the judgment an input and the term an output as shown in Figure 5.6. With the IREFINE rules, we proceed by the shape of the goal type, synthesizing an introduction form I along with refining the examples

$\mathbf{proj}(X) = \overline{\sigma_i \mapsto \chi_{1i}}^{i<n}, \ldots, \overline{\sigma_i \mapsto \chi_{ki}}^{i<n}$
  where $X = \overline{\sigma_i \mapsto C(\chi_{1i}, \ldots, \chi_{ki})}^{i<n}$

$\mathbf{apply}(f, x, \sigma \mapsto pf) = \overline{[pf/f][v_i/x]\sigma \mapsto \chi_i}^{i<m}$
  where $pf = \overline{v_i \Rightarrow \chi_i}^{i<m}$

$\mathbf{distribute}(\Sigma, T, X, E) = (p_1, X'_1), \ldots, (p_m, X'_m)$
  where
    $\mathbf{ctors}(\Sigma, T) = C_1, \ldots, C_m$
    $\forall i \in 1, \ldots, n.\, p_i = \mathbf{pattern}(\Sigma, C_i)$
    $\forall i \in 1, \ldots, n.\, X'_i = [\sigma'\sigma \mapsto \chi \mid \sigma \mapsto \chi \in X, \sigma(E) \longrightarrow^* C_i(\chi 1, \ldots, \chi_k),$
      $\mathbf{vbinders}(p_i, \chi_1, \ldots, \chi_k) = \sigma']$

$\mathbf{ctors}(\Sigma, T) = C_1, \ldots, C_n$
  where $\forall i \in 1, \ldots, n.\, C_i : \tau_1 * \ldots * \tau_k \to T \in \Sigma$

$\mathbf{pattern}(\Sigma, C) = C(x_1, \ldots, x_k)$
  where $C : \tau_1 * \ldots * \tau_k \to T \in \Sigma$

$\mathbf{vbinders}(p, e_1, \ldots, e_k) = [e_1/x_1] \ldots [e_k/x_k]$
  where $p = C(x_1, \ldots, x_k)$

Figure 5.7: ML_{syn} synthesis auxiliary functions

X. As with our presentation of $\lambda^{\rightarrow}_{syn}$, we make this transformation explicit by delegating all of this additional example refinement behavior to meta-functions—**proj**, **apply**, and **distribute** for constructors (IREFINE-BASE), functions (IREFINE-ARR), and matches (IREFINE-MATCH), respectively. The definition of these meta-functions are found in Figure 5.7.

Constructor synthesis generalizes both product and sum synthesis from $\lambda^{\rightarrow}_{syn}$. Whereas products only have two components in $\lambda^{\rightarrow}_{syn}$, constructors have arbitrary, finite arity in ML$_{syn}$. And whereas sums only have two "tags", inl and inr, constructors have an arbitrary, finite amount of tags as dictated by Σ. At base type, we are free to I-refine a particular constructor C with IREFINE-BASE if all the examples share the same head constructor C. To synthesize the ith argument of the constructor expression, we extract the ith argument of each of the examples and use those—along with their respective environments—as our example values. For example, if we had the following collection of example worlds:

$$\begin{aligned} X = \sigma_1 &\mapsto C(\chi_{11}, \chi_{12}, \chi_{13}), \\ \sigma_2 &\mapsto C(\chi_{21}, \chi_{22}, \chi_{23}). \end{aligned}$$

Then IREFINE-BASE would **split** the examples into three collections of example worlds,

$$\begin{aligned} X_1 &= \sigma_1 \mapsto \chi_{11}, \sigma_2 \mapsto \chi_{21} \\ X_2 &= \sigma_1 \mapsto \chi_{12}, \sigma_2 \mapsto \chi_{22} \\ X_3 &= \sigma_1 \mapsto \chi_{13}, \sigma_2 \mapsto \chi_{23}, \end{aligned}$$

which would be used to synthesize the arguments I_1, I_2, and I_3 to C, respectively.

Synthesizing at arrow type is comparatively straightforward. In fact, the IREFINE-ARR rule is almost identical to its $\lambda^{\rightarrow}_{syn}$ variant. The only difference is that with **fix**, we must create a binding for f, the function that we are currently defining or synthesizing. The binding has the goal type, $\tau_1 \rightarrow \tau_2$, and as mentioned previously, we use the example itself—a partial function—as its value.

Note that X is a collection of example worlds where each example world has the form $\sigma \mapsto \chi$. Because of this, we may be refining several example worlds when applying IREFINE-ARR, each of which contains its own partial function value. In $\lambda^{\rightarrow}_{syn}$, refining a single example world versus multiple example worlds using IREFINE-ARR produced the same set of example values and bindings; only the original environments differed. Assuming that the initial environments paired with the example worlds are the same, then the two situations were equivalent.

In ML$_{syn}$, this is no longer the case. Consider the same collection of example

worlds from Section 2.3.4:

$$\begin{aligned} X &= \sigma_1 \mapsto \rho_1, \sigma_2 \mapsto \rho_2 \\ \rho_1 &= v_1 \Rightarrow \chi_1 \mid v_2 \Rightarrow \chi_2 \\ \rho_2 &= v_3 \Rightarrow \chi_3 \mid v_4 \Rightarrow \chi_4 \mid v_5 \Rightarrow \chi_5. \end{aligned}$$

Applying IREFINE-ARR in ML_{syn} produces the following collection of example worlds:

$$\begin{aligned} X' = {} & [\rho_1/f][v_1/x]\sigma_1 \mapsto \chi_1, [\rho_1/f][v_2/x]\sigma_1 \mapsto \chi_2 \\ & [\rho_2/f][v_3/x]\sigma_2 \mapsto \chi_3, [\rho_2/f][v_4/x]\sigma_2 \mapsto \chi_4, [\rho_2/f][v_5/x]\sigma_2 \mapsto \chi_5 \end{aligned}$$

Even if σ_1 and σ_2 are identical, f is bound to either ρ_1 or ρ_2 When evaluating applications to f in these example worlds, we will get different results as ρ_1 and ρ_2 are different partial functions. In particular, we will likely encounter NoMatch exceptions when evaluating one partial function but not the other because they will likely have distinct branches.

In contrast, if we included all the input/output examples in a single partial function:

$$\begin{aligned} X &= \sigma \mapsto \rho \\ \rho &= v_1 \Rightarrow \chi_1 \mid v_2 \Rightarrow \chi_2 \\ &\quad\; v_3 \Rightarrow \chi_3 \mid v_4 \Rightarrow \chi_4 \mid v_5 \Rightarrow \chi_5. \end{aligned}$$

then **apply** produces the following example worlds:

$$\begin{aligned} X' = {} & [\rho/f][v_1/x]\sigma \mapsto \chi_1, [\rho/f][v_2/x]\sigma \mapsto \chi_2 \\ & [\rho/f][v_3/x]\sigma \mapsto \chi_3, [\rho/f][v_4/x]\sigma \mapsto \chi_4, [\rho/f][v_5/x]\sigma \mapsto \chi_5 \end{aligned}$$

where each example world has all of the input/output examples available in the value of f.

Finally, IREFINE-MATCH generalizes the synthesis of sums to data types with an arbitrary number of constructors of arbitrary arity. Like $\lambda^{\rightarrow}_{syn}$, IREFINE-MATCH proceeds in three steps:

1. Guess a value of sum type to pattern match against.
2. Distribute the example worlds among the branches of the pattern match.
3. Recursively synthesize the branches of the pattern match.

The **distribute** takes care of the critical second step, creating m sets of example worlds $X_1, \ldots, X_m$ corresponding to the m possible constructors of the data type. It proceeds by evaluating the scrutinee, E, within each example world of X and

sends that example world to the branch corresponding to the constructor that E evaluates to. Along the way, it binds the appropriate variables and constructor argument values for that branch.

As a concrete example, consider a definition of a bool data type with constructors true and false. That is,

$$\Sigma = \mathsf{true} : \mathsf{bool}, \mathsf{false} : \mathsf{bool}.$$

Then, if we have the following example worlds,

$$X = [\mathsf{true}/b] \mapsto \mathsf{false}, [\mathsf{false}/b] \mapsto \mathsf{true},$$

applying IREFINE-MATCH with b as the scrutinee results in the following two sets of example worlds

$$\begin{aligned} X_1 =& [\mathsf{true}/b] \mapsto \mathsf{false} \\ X_2 =& [\mathsf{false}/b] \mapsto \mathsf{true} \end{aligned}$$

where X_1 are the examples for the true branch of the pattern match (because b evaluates to true in those example worlds) and X_2 are the examples for the false branch of the pattern match (because b evaluates to false).

5.2.3 Structural Recursion in Synthesis

In Chapter 4 we saw that our type-directed synthesis strategy lent itself well to extending the synthesis procedure to new language features. We were able to re-appropriate the type checking rules for a new language feature into synthesis rules. In ML_{syn} we see that these benefits extend beyond language features to any property that a type system may enforce! In particular, we carry over the structural checks (*i.e.*, the **struct** judgment) to ensure that we only synthesize total functions. The **apply** meta-function of IREFINE-ARR annotates the function binding f as recursive (rec) and the argument binding x as an argument of f (arg f). The **distribute** meta-function of IREFINE-MATCH checks to see if an argument of a recursive function is decomposed with a pattern match and annotates any recursive bindings as structurally decreasing (dec f). And finally, EGUESS-APP ensures that for any synthesized application involving a function f marked as rec that its argument is marked as dec f.

Note that the other judgments that we used to ensure totality during type checking are unnecessary, so we omit them in the synthesis judgment. The completeness check **completeness** is implicitly realized because we always synthesize complete pattern matches for a data type. And the positivity restriction on data types is irrelevant because synthesis takes (a well-formed) Σ as a given input.

5.3 Examples in ML_{syn}

Let's explore the expressive power of ML_{syn} with a number of examples. To begin with, data types allow us to directly synthesize programs that we had to encode in $\lambda^{\rightarrow}_{syn}$ and its extensions.

5.3.1 Specification for Multi-argument Functions

First, consider synthesizing the boolean and operator from Section 4.3.1. We require a signature corresponding to the boolean data type:

$$\Sigma = \mathsf{true} : \mathsf{bool}, \mathsf{false} : \mathsf{bool}.$$

Now, we can synthesize a program at goal type $\mathsf{bool} \rightarrow \mathsf{bool} \rightarrow \mathsf{bool}$. Originally, we presented the examples from before as

$$\begin{array}{l} \mathsf{true} \Rightarrow \mathsf{true} \Rightarrow \mathsf{true} \\ \mid \mathsf{false} \Rightarrow \mathsf{true} \Rightarrow \mathsf{false} \\ \mid \mathsf{true} \Rightarrow \mathsf{false} \Rightarrow \mathsf{false} \\ \mid \mathsf{false} \Rightarrow \mathsf{false} \Rightarrow \mathsf{false}, \end{array}$$

a partial function with four input/output examples. An alternative specification groups together similar inputs

$$\begin{array}{r} \mathsf{true} \Rightarrow (\mathsf{true} \Rightarrow \mathsf{true} \mid \mathsf{false} \Rightarrow \mathsf{false}) \\ \mathsf{false} \Rightarrow (\mathsf{true} \Rightarrow \mathsf{false} \mid \mathsf{false} \Rightarrow \mathsf{false}) \end{array}$$

so that we now have a partial function with two outer input/output examples that feature two nested input/output examples each. In $\lambda^{\rightarrow}_{syn}$, there was no semantic distinction between these two presentations so we stuck with the former approach as it appears more natural-looking. However, in ML_{syn}, the second approach is preferred for the reasons described in Section 5.2.2. More specifically, in either case, we will synthesize the following program skeleton:

$$\begin{array}{c} \mathsf{fix}\ f\ (b{:}\mathsf{bool}) : \mathsf{bool} \rightarrow \mathsf{bool} = \\ \mathsf{fix}\ g\ (b{:}\mathsf{bool}) : \mathsf{bool} = \blacksquare. \end{array}$$

Using the second set of examples, we will have four example worlds when synthesizing the body of the inner function. In the first two, g will be bound to the partial function $\mathsf{true} \Rightarrow \mathsf{true} \mid \mathsf{false} \Rightarrow \mathsf{false}$, and in the second two, g will be bound to the partial function $\mathsf{true} \Rightarrow \mathsf{false} \mid \mathsf{false} \Rightarrow \mathsf{false}$. If we use the second set of examples, we will have also four example worlds. However, in one of them, g will be bound to the partial function $\mathsf{true} \Rightarrow \mathsf{true}$, another $\mathsf{true} \Rightarrow \mathsf{false}$, and in the last

two $\mathsf{false} \Rightarrow \mathsf{false}$. In this particular case, we do not need to synthesize a recursive call, so the value assigned to g does not matter. But if it did, then evaluation of g using the second set of examples will likely fail because none of the example worlds received a complete definition of g. Because of this, we always specify multi-argument function examples in a nested style *a la* the second set of examples.

5.3.2 Example: The And Function

With that digression out of the way, let's continue synthesizing the and function using the examples

$$\begin{aligned}\rho = \mathsf{true} \Rightarrow (\mathsf{true} \Rightarrow \mathsf{true} \mid \mathsf{false} \Rightarrow \mathsf{false})\\ \mathsf{false} \Rightarrow (\mathsf{true} \Rightarrow \mathsf{false} \mid \mathsf{false} \Rightarrow \mathsf{false}).\end{aligned}$$

First, we apply IREFINE-ARR twice to refine away the arrows. This produces the following synthesis state:

$$\Gamma \vdash \mathsf{bool} \rhd X \rightsquigarrow \mathsf{fix}\ f\ (b_1{:}\mathsf{bool}) : \mathsf{bool} = \mathsf{fix}\ g\ (b_2{:}\mathsf{bool}) : \mathsf{bool} = \blacksquare$$

where

$$\begin{aligned}\rho_1 &= \mathsf{true} \Rightarrow \mathsf{true} \mid \mathsf{false} \Rightarrow \mathsf{false}\\ \rho_2 &= \mathsf{true} \Rightarrow \mathsf{false} \mid \mathsf{false} \Rightarrow \mathsf{false}\\ \Gamma &= f{:}\mathsf{bool} \to \mathsf{bool} \to \mathsf{bool}\{\mathsf{rec}\}, b_1{:}\mathsf{bool}\{\mathsf{arg}\ f\}\\ &\quad , g{:}\mathsf{bool} \to \mathsf{bool}\{\mathsf{rec}\}, b_2{:}\mathsf{bool} \vdash \mathsf{bool}\\ X &= [\rho/f][\mathsf{true}/b_1][\rho_1/g][\mathsf{true}/b_1] \mapsto \mathsf{true},\\ &\quad [\rho/f][\mathsf{false}/b_1][\rho_2/g][\mathsf{true}/b_1] \mapsto \mathsf{false},\\ &\quad [\rho/f][\mathsf{true}/b_1][\rho_1/g][\mathsf{false}/b_1] \mapsto \mathsf{false},\\ &\quad [\rho/f][\mathsf{false}/b_1][\rho_2/g][\mathsf{false}/b_1] \mapsto \mathsf{false}.\end{aligned}$$

Even though we have recursive functions, we will never be able to use them in this synthesis derivation. This is because we will never be able to manufacture a binding that is structurally decreasing with respect to f or g as neither of the bool constructors contain recursive instances of bool in their types.

Because of this, synthesis proceeds identically to $\lambda^{\rightarrow}_{syn}$ from this point on. We apply IREFINE-MATCH to synthesize a match pattern matching on b_1 with two branches. In the first branch, the true branch, we have to apply IREFINE-MATCH a second time to pattern match over b_2. From there, we can use IREFINE-BASE to synthesize true and false. In the second branch, the false branch, both example world's goal values are false, so we can synthesize false directly using IREFINE-BASE.

5.3.3 Example: The Stutter Function

Next, let's consider synthesizing a recursive function over some inductive data types. Suppose that we have the following signature corresponding to natural numbers nat and lists over natural numbers list,

$$\begin{aligned}\Sigma = \mathsf{O} : \mathsf{nat}, \mathsf{S} : \mathsf{nat} \to \mathsf{nat},\\ \mathsf{Nil} : \mathsf{list}, \mathsf{Cons} : \mathsf{nat} \to \mathsf{list} \to \mathsf{list},\end{aligned}$$

and our goal type is list → list.

First, let's introduce shorthand for these data types. We use numbers in the place of natural numbers, *e.g.*,

$$\begin{aligned}0 &\stackrel{\text{def}}{=} \mathsf{O}\\ 1 &\stackrel{\text{def}}{=} \mathsf{S}(\mathsf{O})\\ 2 &\stackrel{\text{def}}{=} \mathsf{S}(\mathsf{S}(\mathsf{O}))\\ 3 &\stackrel{\text{def}}{=} \mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{O}))).\end{aligned}$$

And we use standard list notation in the place of successive invocations of Nil and Cons, *e.g.*,

$$\begin{aligned}[] &\stackrel{\text{def}}{=} \mathsf{Nil}\\ [0] &\stackrel{\text{def}}{=} \mathsf{Cons}(\mathsf{O}, \mathsf{Nil})\\ [1,0] &\stackrel{\text{def}}{=} \mathsf{Cons}(\mathsf{S}(\mathsf{O}), \mathsf{Cons}(\mathsf{O}, \mathsf{Nil}))\\ [2,1,0] &\stackrel{\text{def}}{=} \mathsf{Cons}(\mathsf{S}(\mathsf{S}(\mathsf{O})), \mathsf{Cons}(\mathsf{S}(\mathsf{O}), \mathsf{Cons}(\mathsf{O}, \mathsf{Nil})))\end{aligned}$$

Now, let's consider the example:

$$\begin{aligned}\rho = [] &\Rightarrow []\\ \mid [0] &\Rightarrow [0,0]\\ \mid [1,0] &\Rightarrow [1,1,0,0]\end{aligned}$$

which is a single partial function with three alternatives. The examples suggest that we should synthesize the standard stutter function that duplicates every element of a list.

First, we apply IREFINE-ARR to refine the goal to a base type. This leaves us with the following synthesis state:

$$f{:}\mathsf{list} \to \mathsf{list}\{\mathsf{rec}\}, l{:}\mathsf{list}\{\mathsf{arg}\, f\} \vdash \mathsf{list} \rhd X \rightsquigarrow \mathsf{fix}\ f\ (l{:}\mathsf{list}) : \mathsf{list} = \blacksquare$$

where

$$\begin{aligned} X = \; & [\rho/f][[]/l] \mapsto [] \\ & [\rho/f][[0]/l] \mapsto [0,0] \\ & [\rho/f][[1,0]/l] \mapsto [1,1,0,0]. \end{aligned}$$

In each of the three resulting example worlds, f is bound to the original partial function ρ and the argument l is bound to each of the left-hand side values of the partial function alternatives.

At this point, we cannot apply IREFINE-BASE because the goal example values do not share the same head constructor (Nil for the first values and Cons for the others). We can try to E-guess with IREFINE-GUESS but no E-term we can generate at this point satisfies all the examples. Therefore, we must apply IREFINE-MATCH to try to make more progress.

The only E-term that we can guess as a scrutinee of the match is l. The list data type has two constructors, so we have to synthesize two branches for the match corresponding to when l is Nil and l is Cons, respectively. Correspondingly, **distribute** creates two sets of example worlds, one for each branch. It sends the first example world to the first branch (because l evaluates to Nil in that branch) and the remaining example worlds to the second branch (because l evaluates to some $\mathsf{Cons}(v_1, v_2)$ in those branches).

Let's consider synthesizing the Nil branch first. The synthesis state in this branch is

$$f{:}\mathsf{list} \to \mathsf{list}\{\mathsf{rec}\}, l{:}\mathsf{list}\{\mathsf{arg}\, f\} \vdash \mathsf{list} \rhd X \rightsquigarrow \blacksquare$$

where

$$X = [\rho/f][[]/l] \mapsto [].$$

This synthesis problem is straightforward to solve. We can either apply IREFINE-BASE to synthesize Nil since our only example value is Nil or IREFINE-GUESS to guess l which evaluates to Nil as desired.

Now, let's turn our attention to the Cons branch. The synthesis state for this branch is:

$$x{:}\mathsf{nat}, l'{:}\mathsf{list}\{\mathsf{dec}\, f\}, f{:}\mathsf{list} \to \mathsf{list}\{\mathsf{rec}\}, l{:}\mathsf{list}\{\mathsf{arg}\, f\} \vdash \mathsf{list} \rhd X \rightsquigarrow \blacksquare$$

where

$$\begin{aligned} X = \; & [0/x][[]/l'][\rho/f][[0]/l] \mapsto [0,0] \\ & [1/x][[0]/l'][\rho/f][[1,0]/l] \mapsto [1,1,0,0]. \end{aligned}$$

First, we can immediately apply IREFINE-BASE because the example goal values share the same head constructor, Cons. This results in two more synthesis sub-goals to fill in the arguments to Cons. The first sub-goal corresponds to the following

refined examples:

$$X = [0/x][[]/l'][\rho/f][[0]/l] \mapsto 0$$
$$\quad [1/x][[0]/l'][\rho/f][[1,0]/l] \mapsto 1.$$

which we can satisfy by applying IREFINE-GUESS and E-guessing x. The second sub-goal corresponds to the following refined examples:

$$X = [0/x][[]/l'][\rho/f][[0]/l] \mapsto [0]$$
$$\quad [1/x][[0]/l'][\rho/f][[1,0]/l] \mapsto [1,0,0].$$

which is identical to the original set of examples except with the head of the example values peeled off. We can apply IREFINE-BASE and IREFINE-GUESS again to arrive at the following partial program for the Cons branch of the pattern match

$$\mathsf{Cons}(x, \mathsf{Cons}(x, \blacksquare))$$

where the final hole has type list and the examples have been refined to:

$$X = [0/x][[]/l'][\rho/f][[0]/l] \mapsto []$$
$$\quad [1/x][[0]/l'][\rho/f][[1,0]/l] \mapsto [0,0].$$

Finally, we can apply IREFINE-GUESS to guess the recursive function call $f\ l'$. To verify that this E-term satisfies the example, we must evaluate $f\ l'$ in each example world:

- In the first world, $l' = []$, so $\rho\,[] \longrightarrow []$ which is identical to the first world's goal value.
- In the second world, $l' = [0]$, so $\rho\,[0] \longrightarrow [0,0]$ which is identical to the second world's goal value.

Because $f\ l'$ satisfies the examples, it is a valid completion of the term, leaving us with the final program:

$$\begin{array}{l}\mathsf{fix}\ f\ (l{:}\mathsf{list}) : \mathsf{list} = \\ \quad \mathsf{match}\ l\ \mathsf{with} \\ \quad \mid\ \mathsf{Nil} \to \mathsf{Nil} \\ \quad \mid\ \mathsf{Cons}(x, l') \to \mathsf{Cons}(x, \mathsf{Cons}(x, f\ l'))\end{array}$$

which is a correct implementation of the desired stutter function.

5.3.4 Trace Completeness

Note that verifying that the recursive call $f\,l'$ when synthesizing stutter depends on the value assigned to f'. Because f' was the stutter function we were synthesizing, we used the partial function ρ that we used as our initial specification of the synthesis problem as f''s value. It turns out our choice of alternatives for ρ was quite convenient as it allowed us to deduce that $f\,l'$ was a satisfying E-term. However, what if we aren't as careful with our specification?

Consider synthesizing the stutter function with this alternative partial function:

$$\begin{aligned}\rho' = {} & [] \Rightarrow [] \\ & \mid [1,0] \Rightarrow [1,1,0,0]\end{aligned}$$

which is identical to ρ except that we removed the alternative $[0] \Rightarrow [0,0]$. With this set of examples, we can, of course, synthesize a simpler function:

$$\begin{aligned}&\mathsf{fix}\ f\ (l{:}\mathsf{list}) : \mathsf{list} = \\ &\quad \mathsf{match}\ l\ \mathsf{with} \\ &\quad \mid\ \mathsf{Nil} \rightarrow [] \\ &\quad \mid\ \mathsf{Cons}(x, l') \rightarrow [1,1,0,0]\end{aligned}$$

which satisfies the examples but doesn't generalize to the stutter function that we want. But, regardless, the stutter function we synthesized previously certainly satisfies these examples as it is a subset of the input/output examples we gave in ρ, so we ought to be able to derive it with ρ'. The synthesis derivation with ρ' proceeds identically to the derivation with ρ until we arrive at determining if $f\,l'$ satisfies the examples. In this case, there is a single example world:

$$X = [1/x][[0]/l'][\rho'/f][[1,0]/l] \mapsto [0,0].$$

However, $\rho'\,[0] \longrightarrow \mathsf{NoMatch}$ because ρ' does not provide an input/output example for $[0]$. According to our compatibility rules, NoMatch is not compatible with anything (including itself), and so we cannot conclude that $f\,l'$ is a satisfying expression.

In general, when we synthesize recursive function calls, for every input/output example we provide, we must provide a corresponding input/output example that describes that function's behavior on an input that is structurally smaller with respect to the original example's input. We call this property of our input/output examples *trace completeness*. For example, consider the input/output examples for ρ above:

- The example $[1,0] \Rightarrow [1,1,0,0]$ requires the example $[0] \Rightarrow [0,0]$ where $[0]$ is structurally smaller than $[1,0]$.
- In turn, the example $[0] \Rightarrow [0,0]$ requires the example $[] \Rightarrow []$ where $[]$ is

structurally smaller than $[0]$.

- Finally, $[]$ is a base case, so the example $[] \Rightarrow []$ does not require any structurally smaller examples.

Note that the input of each additional example that we require is the *largest* term possible that is structurally smaller than the original example's input. Theoretically this is not necessary; we only require a descending chain of input/output examples that satisfies the pattern of decomposition the candidate program takes. In the case of stutter and most other recursive functions, decreasing the size of the input by a single constructor, *i.e.*, what a single match produces, is sufficient. However, other recursive functions may demand different decompositions of the arguments to the function. For example, we may attempt to synthesize a function of the following form

$$\begin{array}{l}\mathsf{fix}\ f\ (l{:}\mathsf{list}) : \tau = \\ \quad \mathsf{match}\ x\ \mathsf{with} \\ \quad \mid\ \mathsf{Cons}(x, l') \rightarrow \\ \quad\quad \mathsf{match}\ l'\ \mathsf{with} \\ \quad\quad\quad \mid\ \mathsf{Cons}(y, l'') \rightarrow \ldots f\ l'' \ldots \\ \quad \ldots\end{array}$$

where we peel away two constructors before making a recursive call.

Our input/output examples need only mimic this behavior. For example, the partial function

$$\begin{array}{l} [] \Rightarrow 0 \\ \mid [1,0] \Rightarrow 1 \\ \mid [3,2,1,0] \Rightarrow 2 \end{array}$$

is trace complete with respect to this recursion pattern, so the above program would satisfy it. However, developing such an example set requires some knowledge about how the eventual implementation of the function behaves, partially defeating the purpose of using a program synthesizer in the first place! Furthermore, we may not be able to avoid specifying other input/output examples even though our desired function has this behavior. For example, if we needed to specify odd-length example lists to synthesize the hypothetical function above, for example, $[2,1,0]$, then we would need to specify the missing example lists of odd-length.

Therefore, in practice, to fulfill trace completeness we always peel away single constructors—in other words, we choose the *largest* structurally decreasing value with respect to an example—when completing the trace. For example, the partial function specification of stutter satisfies this property. In some cases we may be able to get away with fewer examples, but this approach has the benefit of:

1. Completeness with respect to synthesis as this choice ensures that any recursive function call with any decomposition of that call's original argument

will have a corresponding input/output example in the partial function.

2. Requiring that we only know *which* argument of a recursive function will be structurally decreasing which is much more reasonable to assume than the corresponding pattern of recursive calls.

The downside to trace completeness is that it can greatly increase the number of examples needed to synthesize a program, especially when multiple arguments are involved. In Chapter 8, we explore this phenomenon in more detail once we have built up a synthesis procedure from this synthesis judgment.

Trace completeness allows us to overcome the fact that we do not have a complete value available for the recursive function that we are trying to synthesize. However, we typically have part of that function—sometimes even the whole function—already synthesized. For example, when we were synthesizing stutter, the recursive call that we synthesized completed the program, and yet we didn't use this value for the recursive function. If we did, then we wouldn't need our examples to be trace complete because we can simply evaluate the recursive function like normal instead of appealing to a partial function value. The problem is that, in general, this may not be true as we may have only synthesized some of the branches of a pattern match before needing to evaluate a recursive function call. In Chapter 10, we briefly explore how we can use this partial program information to reduce the number of examples we need to provide to the synthesizer.

5.4 Related Work

In this chapter, we extended $\lambda^{\rightarrow}_{syn}$ to synthesize recursive programs and algebraic data types. We now compare our approach to other example-based systems that synthesize recursive functional programs.

5.4.1 Example Rewriting

The earliest example-based synthesis systems such as Thesys [Summers, 1976] took an *analytical* approach to example refinement where the examples were incrementally rewritten to a set of Lisp-like primitives and generalized to create a final program. Modern improvements on Thesys such as Igor2 [Kitzelmann, 2010b] and MagicHaskeller [Katayama, 2012] improve upon Thesys in a variety of ways, in particular, generalizing the rewriting of examples to arbitrary term-rewriting of constructors. This process finds the *least general generalization* of the set of input/output examples through antiunification. This amounts to equating the parts of the examples that are the same and abstracting away the parts that are different as new synthesis sub-problems to solve. For example, for the following

example constructor values:

$$\mathsf{Cons}(0, \mathsf{Nil}) \Rightarrow \mathsf{Cons}(0, \mathsf{Cons}(0, \mathsf{Nil}))$$
$$\mathsf{Cons}(1, \mathsf{Cons}(0, \mathsf{Nil})) \Rightarrow \mathsf{Cons}(1, \mathsf{Cons}(1, \mathsf{Cons}(0, \mathsf{Cons}(0, \mathsf{Nil}))))$$

the least general generalization creates the following constructor value:

$$\mathsf{Cons}(x, l_1) \Rightarrow \mathsf{Cons}(x, \mathsf{Cons}(x, l_2))$$

Where the variable l_2 is a synthesis sub-problem that must be solved.

This process of antiunification describes exactly the behavior of ML$_{syn}$'s I-refinement rules. In particular, you can think of the constructor peeling behavior of the IREFINE-BASE rule as performing one step of anti-unification on the top-most constructor value. Once I-refinement bottoms out, *i.e.*, IREFINE-BASE or IREFINE-ARR no longer apply, we resort to other means to complete the program, similarly to IGOR2 and its related systems.

5.4.2 Examples and Generalization

For ML$_{syn}$, we must utilize E-guessing to complete the program once I-refinement has been exhausted. In the case that we need to synthesize a recursive function call, we require that the examples are trace complete so that evaluation of the recursive call can be performed by the examples, realized now as a partial function. Other example-based synthesis systems also have variants of this requirement. THESYS calls this relationship between examples *differences*, something that IGOR2 also requires. ESCHER [Albarghouthi et al., 2013] also requires a similar sort of property of the examples to evaluate recursive function calls. However, rather than demand this property of the examples upfront, ESCHER queries the user for additional examples as synthesis demands it.

While ML$_{syn}$ and ESCHER both enumerate arbitrary recursive function calls in a generate-and-test style, THESYS and its descendants maintain an analytical approach by recognizing a fixed set of recurrence patterns in the rewritten examples in order to generate recursive calls. The two approaches represent a trade-off in completeness and scalability. The assumption of a fixed set of patterns, similar to a fixed set of combinators [Feser et al., 2015], limits the search space significantly but means that certain patterns of recursive calls cannot be synthesized.

Chapter 6

The Metatheory of ML$_{syn}$

The simple type extensions to $\lambda^{\rightarrow}_{syn}$ that we explored in Chapter 4 did not change $\lambda^{\rightarrow}_{syn}$'s metatheory. In particular, soundness and completeness followed from straightforward lemmas about the behavior of the new language features. However, the introduction of recursion in ML$_{syn}$ is significant enough to reexamine whether the key properties we established in Chapter 3 hold and if not, why we lost them.

6.1 Auxiliary Lemmas

When proving soundness and completeness for $\lambda^{\rightarrow}_{syn}$ and its extensions in Chapter 3 and Chapter 4, we required a number of auxiliary facts for each example-refinement function we introduced:

- Type preservation lemmas stating that the example-refinement function produced well-typed examples.
- Satisfaction soundness lemmas stating that if we had satisfying sub-expressions for examples produced by the example-refinement function, we could constructor an overall satisfying expression for the original examples.
- Satisfaction preservation lemmas stating that if an expression satisfied some examples, then its sub-expressions satisfied the examples produced by the example-refinement function.

We state these lemmas here for ML$_{syn}$.

Lemma 6.1.1 (Type Preservation of apply (ML$_{syn}$)). *If* $\Sigma; \Gamma \vdash \sigma \mapsto \overline{v_i \Rightarrow \chi_i}^{i<m} : \tau_1 \rightarrow \tau_2$ *then* $\Sigma; x{:}\tau_1, \Gamma \vdash \mathbf{apply}(x, \sigma, \overline{v_i \Rightarrow \chi_i}^{i<m}) : \tau_2$.

Lemma 6.1.2 (Type Preservation of Example World Concatenation (ML$_{syn}$)). *If* $\Sigma; \Gamma \vdash X : \tau$ *and* $\Sigma; \Gamma \vdash X' : \tau$ *then* $\Sigma; \Gamma \vdash X +\!\!+ X' : \tau$.

Lemma 6.1.3 (Satisfaction Soundness of apply ($\mathbf{ML}_{syn}$)). *If $I \vDash X'$ then* $\mathsf{fix}\ f\ (x{:}\tau_1) : \tau_2 = I \vDash X$ *where* $X' = \mathbf{apply}(f, x, \sigma_i \mapsto \rho_1) +\!\!+ \ldots +\!\!+ \mathbf{apply}(f, x, \sigma_n \mapsto \rho_n)$.

Lemma 6.1.4 (Type Preservation of proj ($\mathbf{ML}_{syn}$)). *If* $X = \overline{\sigma_i \mapsto C(I_{1i}, \ldots, ki)}^{i<n}$, $C : \tau_1 * \ldots * \tau_k \to T$, $\Sigma; \Gamma \vdash X : T$ *then* $\overline{\Sigma; \Gamma \vdash X_j : \tau_j}^{j<k}$ *where* $\mathbf{proj}(X) = X_1, \ldots X_k$.

Lemma 6.1.5 (Satisfaction Soundness of proj ($\mathbf{ML}_{syn}$)). *If* $\overline{I_j \vDash X_j}^{j<k}$ *then* $C(I_1, \ldots, I_k) \vDash X$ *where* $\mathbf{proj}(X) = X_1, \ldots, X_k$.

Lemma 6.1.6 (Satisfaction Preservation of proj ($\mathbf{ML}_{syn}$)). *If* $C(I_1, \ldots, I_k) \vDash X$ *then* $\overline{I_j \vDash X_j}^{j<k}$ *where* $\mathbf{proj}(X) = X_1, \ldots, X_k$.

Lemma 6.1.7 (Type Preservation of distribute ($\mathbf{ML}_{syn}$)). *If* $\Sigma; \Gamma \vdash X : \tau$ *and* $\Sigma; \Gamma \vdash E \Rightarrow T$, *then* $\overline{\Sigma; \Gamma_i, \Gamma \vdash X_i : \tau}^{i<m}$ *where* $\mathbf{distribute}(\Sigma, T, X, E) = \overline{(p_i, X_i)}^{i<m}$ *and* $\overline{\mathbf{binders}(\Gamma, E, p_i) = \Gamma_i}^{i<m}$.

Lemma 6.1.8 (Satisfaction Soundness of distribute ($\mathbf{ML}_{syn}$)). *Let I be the expression*

$$\begin{array}{l}\mathsf{match}\ E\ \mathsf{with}\\ \mid\ p_1 \to I_1\\ \ldots\\ \mid\ p_m \to I_m.\end{array}$$

If $\overline{I_i \vDash X_i}^{i<m}$ *then* $I \vDash X$ *where* $\mathbf{distribute}(\Sigma, T, X, E) = \overline{(p_i, X_i)}^{i<m}$.

Lemma 6.1.9 (Satisfaction Preservation of distribute ($\mathbf{ML}_{syn}$)). *Let I be the expression*

$$\begin{array}{l}\mathsf{match}\ E\ \mathsf{with}\\ \mid\ p_1 \to I_1\\ \ldots\\ \mid\ p_m \to I_m.\end{array}$$

If $I \vDash X$ then $\overline{I_i \vDash X_i}^{i<m}$ *where* $\mathbf{distribute}(\Sigma, T, X, E) = \overline{(p_i, X_i)}^{i<m}$.

Most of the proofs follow analogously from similar language features we explored in Chapter 3 and Chapter 4, so we do not restate them here. The exceptions to this rule are satisfaction soundness and preservation for **apply**. To see this, let's follow the proof of satisfaction soundness for **apply**:

Consider a single example world $\sigma \mapsto \overline{v_i \Rightarrow \chi_i}^{i<m} \in X$. Unfolding the definition of the satisfies judgment for I shows that:

$$\sigma(I) = \sigma(\mathsf{fix}\ f\ (x{:}\tau_1) : \tau_2 = I_1) \longrightarrow^* \mathsf{fix}\ f\ (x{:}\tau_1) : \tau_2 = \sigma(I_1).$$

Therefore, it suffices to show that $\mathsf{fix}\ f\ (x{:}\tau_1) : \tau_2 = \sigma(I_1) \simeq \overline{v_i \Rightarrow \chi_i}^{i<m}$. By EQ-FIX-PF, this means that we must show that for all $i \in 1, \ldots, m$,

$$I_\sigma\, v_i \longrightarrow [I_\sigma/f][v_i/x]\sigma(I_1) \longrightarrow^* v \wedge v \simeq \chi_i$$

where $I_\sigma = \mathsf{fix}\ f\ (x{:}\tau_1) : \tau_2 = \sigma(I_1)$. However, this follows directly from the fact that $I_1 \vDash X'$ where each example world in X' is of the form $[v_i/x]\sigma \mapsto \chi_i$.

At this point in the proof, we would appeal to the fact that $I_1 \vDash X'$ from our inductive hypothesis. But expanding the definition **apply**, we find that we know that for each $i \in 1, \ldots, m$,

$$[\rho/f][v_i/x]\sigma(I_1) \rightarrow^* v \wedge v \simeq \chi_i.$$

This looks like what we want, but in our goal, we have that the recursive function is substituted for f whereas we know from the inductive hypothesis that ρ is substituted for f instead! However, we know substituting the recursive function for the partial function is sound because, by design of the synthesis algorithm, the recursive function agrees on all the behavior defined by ρ. The reverse direction is more sketchy; we discuss it in Section 6.3.

6.2 Soundness

Recall that soundness of type-directed program synthesis can be broken up into two components:

1. We synthesize well-typed programs.
2. We synthesize programs that satisfy the examples.

Unsurprisingly, both properties hold in ML_{syn}.

Lemma 6.2.1 (Type Soundness of ML_{syn}).

1. *If* $\Gamma \vdash \tau \rightsquigarrow E$, *then* $\Gamma \vdash E \Rightarrow \tau$.
2. *If* $\Gamma \vdash X : \tau$ *and* $\Gamma \vdash \tau \rhd X \rightsquigarrow I$, *then* $\Gamma \vdash I \Leftarrow \tau$.

Proof. By mutual induction on the synthesis derivations for E- and I-terms. Consider the final rule used in the derivation:

Case EGUESS-VAR $E = x$. By the premise of EGUESS-VAR, $x : \tau$ which is sufficient to conclude by T-EVAR that x is well-typed.

Case EGUESS-APP $E = E_1\, I$. By the premises of EGUESS-APP and our inductive hypotheses, we know that E_1 and I are well-typed at $\tau 1 \to \tau$ and $\tau 1$, respectively. With this, we can conclude via T-EAPP that $E1\, I$ is well-typed at type τ.

Case IREFINE-GUESS $I = E$. By the premises of IREFINE-GUESS and our inductive hypothesis, we know that E is well-typed as an E-form and from T-IELIM E is well-typed as an I-form.

Case IREFINE-ARR $I = \mathsf{fix}\ f\ (x{:}\tau_1) : \tau_2 = I_1$. By the premises of IREFINE-ARR and our inductive hypothesis, we know that I_1 is well-typed. Therefore, by T-IARR, we know that I is well-typed.

Case IREFINE-BASE $I = C(I_1, \ldots, I_k)$. By the premises of IREFINE-BASE and our inductive hypothesis, we know that each I_k is well-typed. Therefore, by T-ICTOR, we know that I is well-typed.

Case IREFINE-MATCH $I = \mathsf{match}\ E\ \mathsf{with}\ \overline{p_i \to I_i}^{\,i<m}$. By the premises of IREFINE-MATCH and our inductive hypotheses, we know that each sub-component of the match expression is well-typed. Therefore, by T-IMATCH, we know that I is well-typed. □

Lemma 6.2.2 (Example Soundness of $\mathbf{ML}_{syn}$). *If* $\Gamma \vdash X : \tau$ *and* $\Gamma \vdash \tau \rhd X \rightsquigarrow I$, *then* $I \vDash X$.

Proof. By induction on the synthesis derivation of I. Consider the final rule used in the derivation:

Case IREFINE-GUESS $I = E$. By the premises of IREFINE-GUESS and our inductive hypothesis, we know that E is well-typed as an E-form and from T-IELIM E is well-typed as an I-form.

Case IREFINE-ARR $I = \mathsf{fix}\ f\ (x{:}\tau_1) : \tau_2 = I_1$. By the premises of IREFINE-ARR we know that $X = \sigma_1 \mapsto \rho_1, \ldots, \sigma_n \mapsto \rho_n$ and $X' = \mathsf{apply}(\sigma_1 \mapsto \rho_1) +\!\!+ \ldots +\!\!+ \mathsf{apply}(\sigma_1 \mapsto \rho_n)$. By Lemma 6.1.1 and Lemma 6.1.2, we know that X' is well-typed. This allows us to use our inductive hypothesis to conclude that $I_1 \vDash X'$, and with Lemma 6.1.3, we can conclude our goal.

Case IREFINE-BASE $I = C(I_1, \ldots, I_k)$. By the premises of IREFINE-BASE we know that $X = \overline{\sigma_i \mapsto C(\chi_{i1}, \ldots, \chi_{ik})}^{\,i<n}$ and $\mathsf{proj}(X) = X_1, \ldots, X_k$. By Lemma 6.1.4, we know that the $X_1, \ldots, X_k$ are well-typed. This allows us to use our inductive hypothesis to conclude that $\overline{I_i \vDash X_i}^{\,i<k}$, and with Lemma 6.1.5, we can conclude our goal.

Case IREFINE-MATCH $I = \mathsf{match}\ E\ \mathsf{with}\ \overline{p_i \to I_i}^{i<m}$. By the premises of IREFINE-MATCH we know that we have generated an arbitrary E of type T and $\mathsf{distribute}(\Sigma, T, X, E) = \overline{(p_i, X_i)}^{i<m}$. By Lemma 6.1.7, we know that the $X_1, \ldots, X_m$ are well-typed. This allows us to use our inductive hypothesis to conclude that $\overline{I_i \vDash X_i}^{i<m}$, and with Lemma 6.1.8, we can conclude our goal. □

6.3 Completeness

Soundness ensures that any program that we synthesize is correct with respect to its specification (*i.e.*, types and examples). Completeness ensures that we can synthesize any program expressible in the language. ML_{syn} poses multiple potential difficulties for completeness. The most obvious problem is the presence of recursion. Because the synthesis judgment relies heavily on evaluation—in IREFINE-MATCH when evaluating a scrutinee and in IREFINE-GUESS when normalizing an E-term to perform a compatibility check—synthesis may not terminate on certain inputs if evaluation goes into an infinite loop. Luckily, the ML_{syn} type system employs the same checks as other languages believed to be total to ensure termination (such as Coq [The Coq Development Team, 2012]).

Theorem 6.3.1 (Termination of ML_{syn}). *If $\vdash \Sigma$ and $\Sigma; \Gamma \vdash e : \tau$, then either e is some v or $e \longrightarrow^* v$ and $\Sigma; \Gamma \vdash v : \tau$.*

Because of this theorem (which we do not prove), recursion is not a problem for completeness. However, our use of partial functions as values in ML_{syn} causes dire complications for completeness.

6.3.1 Partial Functions Approximating Recursive Behavior

One problem is related to the proofs of satisfaction soundness and preservation above. We showed that soundness holds, but we needed to reason about where it was safe to substitute a recursive function for a partial function that is its specification. This is ok because we know that the recursive function agrees on all the behavior specified by the partial function. The problem is that satisfaction preservation requires us to reason in the opposite direction: is it ok to substitute a partial function for a recursive function that it specifies? The answer is negative because the partial function will likely to evaluate to NoMatch on inputs that the recursive function would not.

For example, recall the correct implementation of the stutter function that we synthesized earlier:

$$\begin{array}{l}\mathsf{fix}\ f\ (l{:}\mathsf{list}) : \mathsf{list} = \\ \quad \mathsf{match}\ l\ \mathsf{with} \\ \quad\ \mid\ \mathsf{Nil} \to \mathsf{Nil} \\ \quad\ \mid\ \mathsf{Cons}(x, l') \to \mathsf{Cons}(x, \mathsf{Cons}(x, f\, l'))\,.\end{array}$$

Here, it is clear that if we substitute for f the partial function ρ that we used to synthesize stutter,

$$\rho = [] \Rightarrow [] \ \mid\ [0] \Rightarrow [0,0] \ \mid\ [1,0] \Rightarrow [1,1,0,0],$$

we will get NoMatch errors on values not specified by ρ.

6.3.2 Partial Functions as Values

To support partial functions as the program value for recursive functions, we added partial functions to the grammar of plain values. It turns out that this modification alone is sufficient to make ML_{syn} incomplete! This is because as discussed in Section 5.2.1, the evaluation rule for partial functions EVAL-PF-GOOD requires that we compare the inputs of the partial function to the argument of the function call. The compatibility relation from $\lambda^{\to}_{syn}$ is insufficient for this purpose because it compares an example value χ and a value v. We therefore extend the compatibility relation to compare two values, taking advantage of the fact that because we needed to extend values with partial functions, example values are now a proper subset of plain values.

Because the compatibility relation now compares values, we must contend with comparing functions for equality. This arises when evaluating higher-order partial functions. For example, if we evaluate $(v_1 \Rightarrow \chi)\, v_2$, then we may compare two function values for equality if v_1 and v_2 are fixes. In $\lambda^{\to}_{syn}$, $\beta\eta$-equivalence over functions is decidable. However, in the presence of recursion, equivalence of functions is no longer decidable. To see this, note that ML_{syn} is at least as expressive as the primitive recursive functions whose equality is known to be undecidable [Kahrs].

Therefore, we must resort to EQ-REFL that only equates two function values if they are syntactically identical (up to renaming of bound variables). However, this approximation means that we cannot solve some higher-order synthesis problems. For example, consider synthesizing at the higher goal type $((\mathsf{nat} \to \mathsf{nat}) \to \mathsf{nat}) \to \mathsf{nat}$. Now suppose that we have a partial function with a single input/output example

$$(\lambda x{:}\mathsf{nat}.\, x \Rightarrow \mathsf{O}) \Rightarrow \mathsf{O}$$

and synthesize in a context

$$\Gamma = \mathsf{plus} : \mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}, \mathsf{id} : \mathsf{nat} \to \mathsf{nat}.$$

In our starting environment, plus is bound to the usual definition of addition over natural numbers and id is bound to the value $\lambda x{:}\mathsf{nat}.\ \mathsf{plus}\ x\ \mathsf{O}$. Then, we are unable to synthesize the candidate function:

$$\lambda f{:}(\mathsf{nat} \to \mathsf{nat}) \to \mathsf{nat}.\ f\ \mathsf{id}.$$

To see why, note that we must evaluate the body of the function, $f\ \mathsf{id}$, when applying IREFINE-GUESS. Evaluation results in NoMatch because the functions $\lambda x{:}\mathsf{nat}.\ \mathsf{plus}\ x\ \mathsf{O}$ and $\lambda x{:}\mathsf{nat}.\ x$ are not syntactically identical.

Chapter 7

Implementation

So far, we have explored the theoretical foundations of program synthesis with types in detail by developing and extending core synthesis calculi equipped with increasingly complex types. Now that we have a core calculus that closely resembles a subset of a real-world functional programming language, ML_{syn}, we now turn our attention to deriving a synthesis procedure from our synthesis calculus and then making that procedure efficient.

7.1 Synthesis Trees

Given a specification—a goal type and examples—the possible synthesis derivations form a tree of possible satisfying programs, a *synthesis tree*. For example, consider synthesizing a function of goal type $\mathsf{bool} \to \mathsf{bool} \to \mathsf{bool}$ from the partial function with a single input/output example

$$\mathsf{true} \Rightarrow \mathsf{true} \Rightarrow \mathsf{true}.$$

There are many programs that fulfill this specification, for example, the constant true function

$$\lambda b_1{:}\mathsf{bool}.\, \lambda b_2{:}\mathsf{bool}.\, \mathsf{true},$$

the left-select function

$$\lambda b_1{:}\mathsf{bool}.\, \lambda b_2{:}\mathsf{bool}.\, b_1,$$

the right-select function,

$$\lambda b_1{:}\mathsf{bool}.\, \lambda b_2{:}\mathsf{bool}.\, b_2,$$

and the and operator,

$$\begin{array}{l}\lambda b_1{:}\mathsf{bool}.\, \lambda b_2{:}\mathsf{bool}.\, \mathsf{match}\, b_1\, \mathsf{with}\\ \quad \mid\ \mathsf{true} \to b_2\\ \quad \mid\ \mathsf{false} \to \mathsf{false}.\end{array}$$

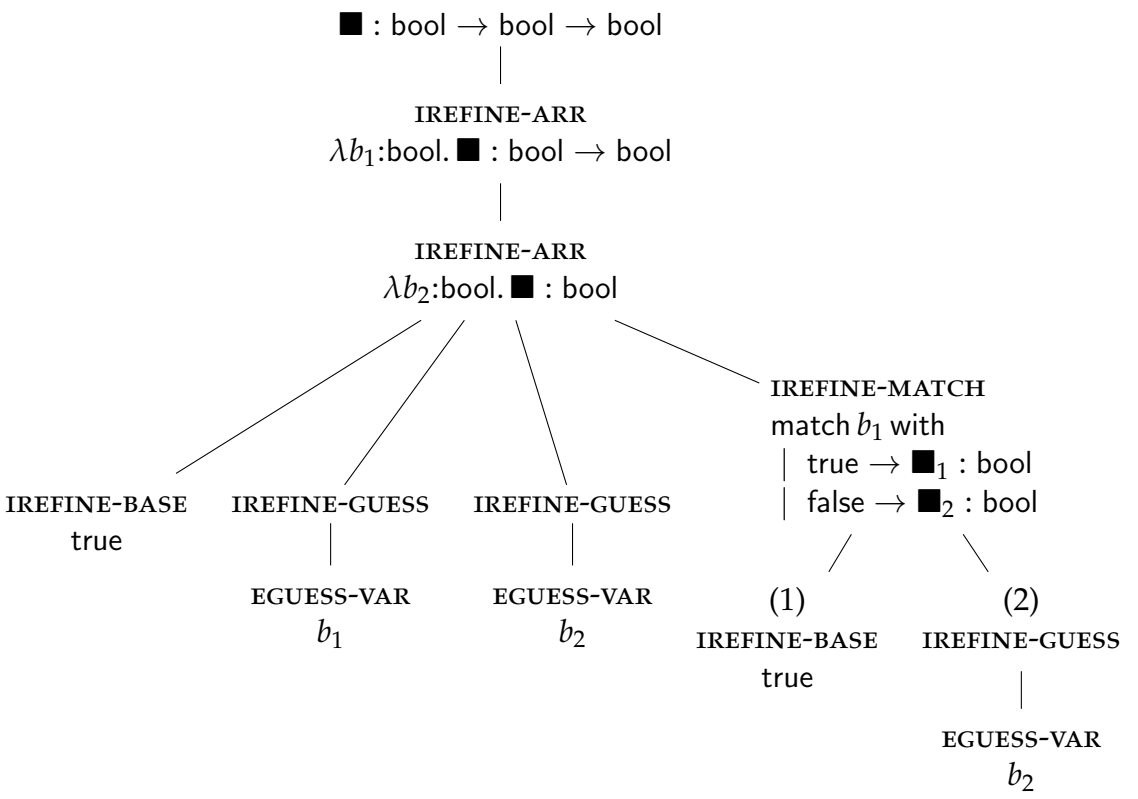
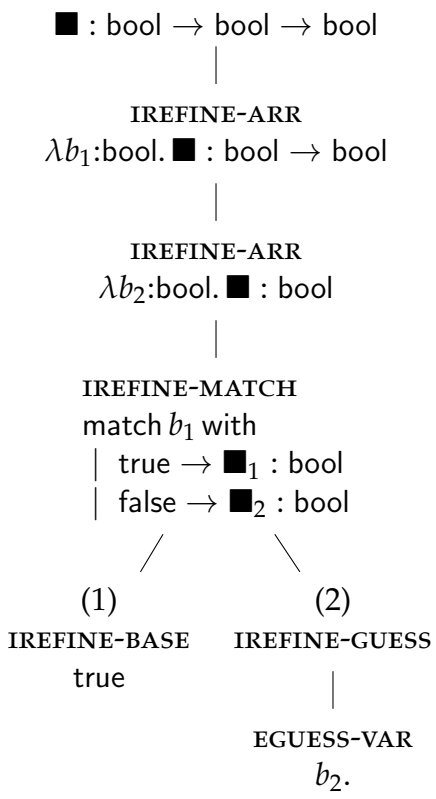


Figure 7.1: Example synthesis tree.

We give the synthesis tree corresponding to these potential derivations in Figure 7.1. Nodes in this tree correspond to potential application of rules from our synthesis judgment, and edges connects the rule applications to the sub-goals, denoted by holes ■, that they fulfill. The root of the tree, the single *hole*, corresponds to our initial synthesis goal. In some cases, a single sub-goal is left behind after applying a rule, for example, when applying IREFINE-ARR, so the children of that node represent the different ways we can complete that sub-goal. In other cases, multiple sub-goals are left behind, for example, when applying IREFINE-MATCH. When this happens, we annotate the corresponding child nodes so it is clear which sub-goal they are meant fulfill.

When we talked about synthesizing a program in earlier chapters, we have chosen a particular derivation out of this tree. For example, the derivation corresponding to the and function that we synthesized in Chapter 4 is given by:



7.2 Collection Semantics

Now that we are concerned with implementation, it is no longer sufficient to merely assert that a single derivation or path exists in this synthesis tree. We must, instead, derive a synthesis procedure that constructs a refinement tree and extracts a derivation from that tree which corresponds to a satisfying program. Note that it may be useful to construct more of a refinement tree than a single derivation. For example, it may be more efficient to derive parts of the refinement tree first, leaving more time-consuming portions of the tree for later as we explore in Section 7.3.3. We may also want to be able to choose from among multiple satisfying programs using additional heuristics, for example, the number of variables or external functions used. Or, we may want to display the list of potential programs to the user and let them choose.

For the purposes of our work, we'll restrict ourselves to generating a single satisfying program. To build a synthesis procedure, we observe that we can classify the synthesis rules of ML_{syn} into three sorts:

1. *Refinement* rules that allow us to act in a type-directed manner by refining a synthesis goal according to the goal type and examples,

2. *Guessing* rules that allow us to guess an E-term and check it against the examples, and

3. *Matching* which allows us to generate more information by performing case analysis on a value.

The refinement rules, IREFINE-ARR and IREFINE-BASE apply at distinct types, so there is never any question as to which refinement rule to apply at a particular synthesis state. Guessing via IREFINE-GUESS, on the other hand, may apply at any time whenever we can generate an E-term from the context of the appropriate type. Likewise, IREFINE-MATCH also applies whenever we can generate an E-term of base type to pattern match over.

We can explore these possibilities by using an iterative deepening approach by the size of the derivation. First let's define a *collection semantics* derived from our synthesis rules. Let the function $\mathbf{refine}(\Sigma; \Gamma; \tau; X; k)$ produce the set of all programs of goal type τ that satisfy the examples X that are exactly of size k (as determined by the number of nodes in their abstract syntax trees). With this function, we can define a simple synthesis procedure over an initial synthesis state $(\Sigma, \Gamma, \tau, X)$ as follows:

1. Let $k = 1$ initially.

2. Calculate $\mathcal{E} = \mathbf{refine}(\Sigma; \Gamma; \tau; X; k)$.

   - If $\mathcal{E} \neq \emptyset$, then return any program from $\mathcal{E}$.
   - If $\mathcal{E} = \emptyset$, then increment k and repeat step (2).

Figure 7.2 and Figure 7.3 give the definition of these semantics, appealing to the auxiliary functions and definitions for synthesis we defined earlier in Figure 5.6 and Figure 5.7. This collection of functions improves on the synthesis judgments of ML_{syn} in a number of ways:

1. The non-deterministic choice of application of the rules at each synthesis step is made explicit. In particular, with each call of **refine**, we calculate the set of programs we would generate if we refined by type (**refine**), guessed E-terms (**guess**), or pattern matched (**match**) and take the union of them as the final result.

2. By placing an upper bound on the size of the synthesized programs, individual calls to **refine** always terminate as the size of the desired programs always decreases with each successive call to **refine** or one of its helper functions.

3. All implicitly quantified variables are made explicit. For example, **match** implicitly quantifies over all base types, so we take the union of performing pattern matching over each of these types.

$$\mathbf{guess}(\Sigma;\Gamma;\tau;X;0) = \{\}$$
$$\mathbf{guess}(\Sigma;\Gamma;\tau;X;k) = \{E \mid E \in \mathbf{gen}(\Sigma;\Gamma;\tau;k), E \vDash X\}$$

$$\mathbf{types}(\Sigma) = \{T \mid C : \tau_1 * \ldots * \tau_m \to T \in \Sigma\}$$
$$\mathbf{match}(\Sigma;\Gamma;\tau;X;0) = \{\}$$
$$\mathbf{match}(\Sigma;\Gamma;\tau;X;k) = \bigcup_{T \in \mathbf{types}(\Sigma)} \bigcup_{\substack{k',k_1,\ldots,k_m \text{ for} \\ 1+k'+k_1+\ldots+k_m=k}}$$
$$\{\mathsf{match}\ E\ \mathsf{with}\ \overline{p_i \to I_i}^{i<m} \mid$$
$$E \in \mathbf{gen}(\Sigma;\Gamma;T;k'),$$
$$\overline{(p_i, X_i)}^{i<m} = \mathbf{distribute}(\Sigma, T, X, E),$$
$$\overline{\Gamma_i}^{i<m} = \mathbf{binders}(\Gamma, E, p_i),$$
$$\overline{I_i \in \mathbf{refine}(\Sigma;\Gamma_i,\Gamma;\tau;X_i;k_i)}^{i<m}\}$$

$$\mathbf{refine}(\Sigma;\Gamma;\tau;X;0) = \{\}$$

$$\mathbf{refine}(\Sigma;\Gamma;\tau_1 \to \tau_2;X;k) = \{\mathsf{fix}\ f\ (x{:}\tau_1) : \tau_2 = I \mid$$
$$X' = \mathbf{apply}(f, x, \sigma_1 \mapsto \rho_1) +\!\!+ \ldots +\!\!+ \mathbf{apply}(f, x, \sigma_n \mapsto \rho_n),$$
$$I \in \mathbf{refine}(\Sigma; f{:}\tau_1 \to \tau_2\{\mathsf{rec}\}, x{:}\tau_1\{\mathsf{arg}\ f\};\Gamma;X';k-1)$$
$$\} \cup \mathbf{guess}(\Sigma;\Gamma;\tau;X;k) \cup \mathbf{match}(\Sigma;\Gamma;\tau;X,k)$$
where
$$X = \sigma_1 \mapsto \rho_1, \ldots, \sigma_n \mapsto \rho_n$$

$$\mathbf{refine}(\Sigma;\Gamma;T;X;k) = \bigcup_{\substack{k_1,\ldots,k_m \text{ for} \\ 1+k_1+\ldots+k_m=k}}$$
$$\{C(I_1,\ldots,I_m) \mid$$
$$X_1,\ldots,X_m = \mathbf{proj}(X),$$
$$\overline{I_i \in \mathbf{refine}(\Sigma;\Gamma;\tau_i;X_i;k_i)}^{i<m}$$
$$\} \cup \mathbf{guess}(\Sigma;\Gamma;\tau;X;k) \cup \mathbf{match}(\Sigma;\Gamma;\tau;X;k)$$
where
$$X = \overline{\sigma_i \mapsto C(I_{i1},\ldots,C_{im})}^{i<n}$$
$$C : \tau_1 * \ldots * \tau_m \in \Sigma$$

Figure 7.2: ML_{syn} collection semantics

$$\begin{aligned}
&\mathbf{gen}(\Sigma;\Gamma;\tau;0) = \{\} \\
&\mathbf{gen}(\Sigma;\Gamma;\tau;1) = \{x \mid x{:}\tau \in \Gamma\} \\
&\mathbf{gen}(\Sigma;\Gamma;\tau;k) = \bigcup_{x:\tau_1\to\tau\in\Gamma} \bigcup_{\substack{k_1,k_2 \text{ for} \\ 1+k_1+k_2=k}} \\
&\quad \{E\ I \mid E \in \mathbf{gen}(\Sigma;\Gamma;\tau_1 \to \tau;k_1), I \in \mathbf{refine}(\Sigma;\Gamma;\tau_1;\cdot;k_2), \mathbf{struct}(\Gamma,E,I)\}
\end{aligned}$$

Figure 7.3: ML_{syn} term generation

These changes result in a synthesis procedure that is a straightforward translation of our original synthesis judgment.

The three main synthesis operations that we identified are divided up into three functions: **guess**, **match**, and **refine**. **guess** implements the behavior of the IREFINE-GUESS rule, generating the set of E-terms of size k that satisfy the examples, written $E \vDash X$. The satisfaction check, reproduced below from Chapter 5

$$\forall \sigma \mapsto \chi \in X.\, \sigma(E) \longrightarrow^* v \wedge v \simeq \chi.$$

requires that, within each example world, the synthesized program evaluates to a value that is compatible the goal value for that world. **guess** generates candidate E-terms through the **gen** helper function which performs simple raw-term enumeration over a particular type. When enumerating applications, the argument may be an I-term according to the grammar of ML_{syn}. To generate I-terms, we appeal back to **refine** but pass in the empty set of examples. Like the synthesis judgments from which it is derived, **refine** degenerates into raw-term enumeration when given no examples.

match corresponds to invocations of IREFINE-MATCH that generate matches for all possible scrutinee base types at a particular goal type. For a particular base type, we generate sets of expressions for all of the components of the match—the scrutinee and each of the branches—and form matches by taking the Cartesian product of these sets. To create match expressions of size k, we also consider all the ways that we can distribute the size among these match components. Note that if any of the sets of expressions are empty—*i.e.*, we were not able to generate any scrutinees of the given base type or a satisfying expression for a particular branch—we do not produce a match expression for the base type that we are pattern matching over.

Finally, **refine** performs type-directed program and example refinement. At arrow type, we perform refinement according to IREFINE-ARR, and at base type when the examples share the same head constructor, we perform refinement according to IREFINE-BASE. In either case, we also invoke **guess** and **match** as discussed earlier to capture the full set of possible synthesis derivations.

7.2.1 The Minimum Program Principle

The **refine** function is governed by the size of the derivation k. Because each rule application adds one abstract syntax tree (AST) node to the size of the synthesized programs, k also corresponds to program size. Therefore, the above procedure produces the smallest program possible that satisfies the input examples. However, is this a desirable property of a program synthesizer? The following principle answers this question in the affirmative:

Definition 7.2.1 (The Minimum Program Principle). *In program synthesis, smaller satisfying programs (in terms of program size) are more likely to generalize to the desired behavior intended by the user.*

The Minimum Program Principle is not a provable theorem, but a search heuristic inspired by Occam's razor that many program synthesis tools exploit [Albarghouthi et al., 2013; Feser et al., 2015; Perelman et al., 2014; Summers, 1976] to refine the space of programs even further.

Intuitively, the Minimum Program Principle observes that a smaller program is less likely to over-specialize on the examples given to the synthesizer. To see this, consider the specification we gave for stutter in Chapter 5:

$$\begin{array}{l} [] \Rightarrow [] \\ \mid [0] \Rightarrow [0,0] \\ \mid [1,0] \Rightarrow [1,1,0,0] \end{array}$$

The desired function that we wanted to synthesize was

$$\begin{array}{l} \mathsf{fix}\ f\ (l{:}\mathsf{list}) : \mathsf{list} = \\ \quad \mathsf{match}\ l\ \mathsf{with} \\ \quad \mid\ \mathsf{Nil} \rightarrow \mathsf{Nil} \\ \quad \mid\ \mathsf{Cons}(x, l') \rightarrow \mathsf{Cons}(x, \mathsf{Cons}(x, f\ l')), \end{array}$$

and we demonstrated that there is a synthesis derivation of this program in ML_{syn}. However, this is not the only synthesis derivation possible. For example, the following satisfying program is derivable in ML_{syn}:

$$\begin{array}{l} \mathsf{fix}\ f\ (l{:}\mathsf{list}) : \mathsf{list} = \\ \quad \mathsf{match}\ l\ \mathsf{with} \\ \quad \mid\ \mathsf{Nil} \rightarrow [] \\ \quad \mid\ \mathsf{Cons}(x, l') \rightarrow \mathsf{match}\ l'\ \mathsf{with} \\ \quad\quad \mid\ \mathsf{Nil} \rightarrow [0,0] \\ \quad\quad \mid\ \mathsf{Cons}(y, l'') \rightarrow \mathsf{match}\ l''\ \mathsf{with} \\ \quad\quad\quad \mid\ \mathsf{Nil} \rightarrow [1,1,0,0] \\ \quad\quad\quad \mid\ \mathsf{Cons}(z, l''') \rightarrow []. \end{array}$$

which is the program that pattern matches repeatedly looking for the inputs specified by the partial function, produces the corresponding outputs, and produces an arbitrary value when the input is not specified.

This second program is less desirable than the first because it overspecializes on the given examples. While it satisfies those particular examples, it does not generalize appropriately to other cases. Note that the overspecializing program is necessarily large because it must use repeated pattern matches and literal values to reproduce the behavior of the partial function. By our size metric of counting AST nodes, this overspecialized program has size 25 whereas the desired program only has size 11. In contrast, a smaller program is likely to use more E-terms—variables and applications—to satisfy the examples because they require less size to accomplish more varied behavior.

7.2.2 Restricting the Search Space with Examples

Now that we have our synthesis procedure, it now makes sense to talk about the effect of examples on the output of the synthesis process. In the previous section, we saw that the examples that we have used to synthesize stutter admit multiple programs. In particular, we were able to derive the standard implementation of stutter along with a program that overspecialized on the examples. Consider a simpler example: synthesizing a program at goal type list $\to$ list with the example

$$[] \Rightarrow [].$$

Certainly, we can synthesize both the implementation of stutter and the overspecialized program from this single example. However, this simple partial function example admits many more programs, for example, the constant [] function λl:list. [] and the identity function λl:list. l. Because our synthesis procedure favors smaller programs, we will either produce the constant or identity functions from this example.

From this, we can see that the effect of adding examples is to rule out these simpler programs from consideration. In the case where we have all three examples for stutter, the constant and identity functions are no longer satisfying programs. Suppose that we instead provide only two input/output examples for stutter:

$$\begin{aligned} &[] \Rightarrow [] \\ \mid\ &[0] \Rightarrow [0, 0]. \end{aligned}$$

Is this sufficient to synthesize stutter with our synthesis procedure? It turns out

the answer is no as the procedure produces the following program:

$$
\begin{array}{l}
\mathsf{fix}\ f\ (l{:}\mathsf{list}) : \mathsf{list} = \\
\quad \mathsf{match}\ l\ \mathsf{with} \\
\quad\ |\ \mathsf{Nil} \to \mathsf{Nil} \\
\quad\ |\ \mathsf{Cons}(x, l') \to \mathsf{Cons}(x, l).
\end{array}
$$

This program is smaller than the desired stutter—size 7 versus size 11—so our synthesis procedure would choose it first. Thus it turns out that all three examples are necessary for our procedure to rule out enough smaller programs to produce stutter.

What happens if we include more input/output examples with the caveat that these new examples still imply the stutter function and preserve trace completeness? For example, consider the partial function

$$
\begin{array}{l}
\quad [] \Rightarrow [] \\
| \ [0] \Rightarrow [0, 0] \\
| \ [1, 0] \Rightarrow [1, 1, 0, 0] \\
| \ [2, 1, 0] \Rightarrow [2, 2, 1, 1, 0, 0] \\
| \ [3, 2, 1, 0] \Rightarrow [3, 3, 2, 2, 1, 1, 0, 0]
\end{array}
$$

which contains two additional input/output examples over the previous case. The program corresponding to the overspecialization of the examples is substantially larger at 76 AST nodes.

$$
\begin{array}{l}
\mathsf{fix}\ f\ (l{:}\mathsf{list}) : \mathsf{list} = \\
\quad \mathsf{match}\ l\ \mathsf{with} \\
\quad\ |\ \mathsf{Nil} \to [] \\
\quad\ |\ \mathsf{Cons}(v, l1) \to \mathsf{match}\ l1\ \mathsf{with} \\
\quad\quad\ |\ \mathsf{Nil} \to [0, 0] \\
\quad\quad\ |\ \mathsf{Cons}(w, l2) \to \mathsf{match}\ l2\ \mathsf{with} \\
\quad\quad\quad\ |\ \mathsf{Nil} \to [1, 1, 0, 0] \\
\quad\quad\quad\ |\ \mathsf{Cons}(x, l3) \to \mathsf{match}\ l3\ \mathsf{with} \\
\quad\quad\quad\quad\ |\ \mathsf{Nil} \to [2, 2, 1, 1, 0, 0] \\
\quad\quad\quad\quad\ |\ \mathsf{Cons}(y, l4) \to \mathsf{match}\ l4\ \mathsf{with} \\
\quad\quad\quad\quad\quad\ |\ \mathsf{Nil} \to [3, 3, 2, 2, 1, 1, 0, 0] \\
\quad\quad\quad\quad\quad\ |\ \mathsf{Cons}(z, l5) \to [].
\end{array}
$$

However, by definition the desired implementation of stutter satisfies these additional input/output examples. Furthermore, we know the original examples rule out all programs smaller than the desired program. So these additional examples neither change the output of our synthesis procedure nor rule out additional programs that we would have considered during the synthesis process. In fact,

these additional examples only add additional overhead (a slight amount, as we discuss in Chapter 8) to the synthesis procedure as we must refine and check for satisfaction against more example worlds.

Therefore, there is a balance to providing examples to our synthesis procedure. We must provide enough examples to rule out smaller, trivial programs while satisfying trace completeness. But we should avoid providing too many examples as extra examples do not contribute to the synthesis process. In Chapter 8, we discuss our experience developing examples sets for this synthesis procedure with these considerations in mind.

7.3 Optimizing The Synthesis Procedure

Now that we have defined our synthesis procedure, we turn our attention towards optimizing it. We have already pruned the search space significantly by considering only normal forms and limiting ourselves to structural recursion. Now, we introduce additional techniques inspired from logic and the proof search literature to further optimize our search.

7.3.1 Invertible Rules

One trouble we have with our synthesis judgment and consequently our synthesis procedure is the order in which we invoke particular rules. For example, suppose that we are synthesizing at base type and we have the opportunity to either synthesize a constructor value or a pattern match. Does the order in which we invoke the rules matter? Certainly it seems undesirable to have pattern matches as the arguments to the constructor value, but you can imagine scenarios in which this might occur in the desired program. We reflect this possibility by always invoking the various synthesis functions at every step when possible. But we can clearly save ourselves some work if we can rule out some of these possibilities, preferably without losing completeness.

One such optimization along these lines concerns *invertible* rules in logic. An invertible rule is one where the premises of the rule are derivable whenever the conclusion is. For example, consider IREFINE-ARR, reproduced below:

IREFINE-ARR

$$\frac{\begin{array}{c} X = \sigma_1 \mapsto pf_1, \ldots, \sigma_n \mapsto pf_n \\ X' = \mathbf{apply}(f, x, \sigma_1 \mapsto pf_1) +\!\!+ \ldots +\!\!+ \mathbf{apply}(f, x, \sigma_n \mapsto pf_n) \\ \Sigma; f{:}\tau_1 \to \tau_2\{\mathsf{rec}\}, x{:}\tau_1\{\mathsf{arg}\, f\}, \Gamma \vdash I \rhd X' \rightsquigarrow \tau_2 \end{array}}{\Sigma; \Gamma \vdash \tau_1 \to \tau_2 \rhd X \rightsquigarrow \mathsf{fix}\ f\ (x{:}\tau_1) : \tau_2 = I}.$$

Invertibility states that if we are able to synthesize a program at type $\tau_1 \to \tau_2$ that satisfies X, then we are able to synthesize an I at type τ_2 that satisfies X'.

The usual interpretation of the inference rule makes the opposite claim—if we can synthesize I, then we can synthesize a fix at type $\tau_1 \to \tau_2$. This is true for IREFINE-ARR because of η-expansion; if we have a program I of type $\tau_1 \to \tau_2$, it is equivalent to its η-expansion: $\mathsf{fix}\ f\ (x{:}\tau_1) : \tau_2 = I\ x$.

Because of this property, we can always apply IREFINE-ARR first when searching for a program until it no longer applies without loss of completeness. IREFINE-ARR acts in a type-directed manner over arrow types which are finite in length, so this process always terminates. In our collection semantics, we change **refine** to only perform program-and-example refinement at arrows:

$$
\begin{aligned}
&\mathbf{refine}(\Sigma; \Gamma; \tau_1 \to \tau_2; X; k) = \{\mathsf{fix}\ f\ (x{:}\tau_1) : \tau_2 = I \mid \\
&\quad X' = \mathbf{apply}(f, x, \sigma_1 \mapsto \rho_1) +\!\!+ \ldots +\!\!+ \mathbf{apply}(f, x, \sigma_n \mapsto \rho_n), \\
&\quad I \in \mathbf{refine}(\Sigma; f{:}\tau_1 \to \tau_2\{\mathsf{rec}\}, x{:}\tau_1\{\mathsf{arg}\, f\}; \Gamma; X'; k-1) \\
&\} \\
&\quad \text{where} \\
&\qquad X = \sigma_1 \mapsto \rho_1, \ldots, \sigma_n \mapsto \rho_n.
\end{aligned}
$$

Note that we no longer need to calculate **guess** and **match** at arrow types, saving us a significant amount of work.

With this optimization, we will always synthesize programs that are η-long in addition to β-normal. Concretely, this means that if we are producing a program of type $\tau_1 \to \tau_2$ and can synthesize some non-fix expression I of that type to fulfill the goal, our synthesis procedure will produce its η-expansion rather than it directly. This results in slightly more verbose programs but prunes the search space significantly.

7.3.2 Reigning in Matches

In all program synthesis systems, the conditional proves to be most tricky to generate efficiently. This is because conditionals only *add* information to our synthesis problem, increasing the space of possible programs. In the case of pattern matching, it is clear that this information comes in the form of additional variables into the context. However, with plain old if-expressions,

$$\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else},$$

the information is implicit; we get to assume that e_1 is true in the first branch and false in the second branch. In either case, the difficulty is choosing an appropriate match or conditional scrutinee to make progress. The problem is that there is no indication that our choice will ultimately help us satisfy our goal! As a result, we need to explore each such scrutinee to completion because we don't know if it will result in a satisfying program.

These problems clearly manifest themselves in our synthesis procedure. While

we invoke **match** only at base types thanks to our invertibility optimization from Section 7.3.1, **match** still chooses arbitrary scrutinees generated via **gen**. For a given target program size k, there are, thankfully, a finite number of scrutinees so a call to **match** terminates. However, the number of scrutinees scales exponentially with k as well as the size of the context. Therefore, to remain tractable, we need some additional heuristics to keep the number of scrutinees manageable.

Informativeness In general, we cannot tell upfront if a particular match scrutinee will allow us to create a satisfying program. However, we can apply heuristics to prune out scrutinees that are unlikely to help us to make progress. Some of these heuristics weaken completeness further; we will be unable to synthesis some programs given example sets that the programs otherwise satisfy. However, in return we can make synthesis with match expressions much more efficient.

Recall that when we synthesize a match, we distribute the example worlds to the branches of the match according to how the match scrutinee evaluates in each world. For example, when we synthesized stutter, we had the three input/output examples:

$$\begin{aligned} &[] \Rightarrow [] \\ \mid\ &[0] \Rightarrow [0,0] \\ \mid\ &[1,0] \Rightarrow [1,1,0,0]. \end{aligned}$$

When we synthesized a match expression of the form:

$$\begin{aligned} &\mathsf{match}\ l\ \mathsf{with} \\ &\mid\ \mathsf{Nil} \rightarrow \blacksquare \\ &\mid\ \mathsf{Cons}(x, l') \rightarrow \blacksquare \end{aligned}$$

We sent the first example world to the Nil branch (because $l = []$ in that world) and the remaining two example worlds to the Cons branch (because $l = \mathsf{Cons}(\ldots)$ in those worlds). Because at least one example was sent into each branch, we were able to employ type-and-example-directed guidance to complete the program.

However, in general, this may not be the case. We may end up in situations, for example, where we synthesize a m-way pattern match:

$$\begin{aligned} &\mathsf{match}\ e\ \mathsf{with} \\ &\mid\ C_1\ p_1 \rightarrow \blacksquare \\ &\ldots \\ &\mid\ C_m\ p_m \rightarrow \blacksquare \end{aligned}$$

and the examples are distributed in such a way that some branches do not receive any examples. This is not a problem theoretically as our collection semantics degenerates to raw term enumeration in the absence of examples. However,

because any well-typed expression is a valid completion for these branches with no examples (the satisfaction condition is trivially satisfied), we will likely choose a small expression such as a single variable or a constant that will not generalize to the behavior that the user intends with the examples they provide.

Worst yet, imagine if all of the examples were sent to a single branch.[1] An expression for that branch would need to satisfy all of the original examples with only the additional information provided by the binders introduced by the pattern of the match for support. All of the remaining branches would need to be generated using raw-term enumeration. In this situation, we have done little to refine the synthesis state as we have not taken advantage of the case analysis that the pattern match provides. As a result, it is unlikely that this pattern match will actually help us in generating a final, satisfying program.

We call the likelihood that a pattern match contributes to a satisfying program its *informativeness*. The shape of a particular pattern match alone does not tell us much about its informativeness, but our examples gives us the additional insight we need. Intuitively, an informative pattern match is one that distributes the examples evenly among the branches of the pattern match. In contrast, an uninformative pattern match distributes most of the examples to a minority of the branches. In the worst case, a pattern match distributes all of the examples to a single branch which signifies that we did not "learn" anything about the examples by pattern matching on the given scrutinee.

Elaborate heuristics are possible where we prioritize the exploration particular pattern matches based on their informativeness. For our purposes, we implement a simple heuristic to rule out this worst case scenario for informativeness.

Definition 7.3.1 (Informativeness Restriction (A)). *A pattern match is valid if whenever the pattern match is over a data type with more than one constructor that* distribute *sends examples to at least two distinct branches.*

This restriction, applied to the match function from Figure 7.2 is sufficient to rule out the worst case pattern matches described above. However, it still allows for branches to be synthesized in the absence of examples. The following version of the informativeness restriction fixes this problem.

Definition 7.3.2 (Informativeness Restriction (B)). *A pattern match is valid if whenever the pattern match is over a data type with more than one constructor that* distribute *sends at least one example to each branch of the match.*

The two restrictions represent a trade-off between performance and completeness. Restriction (A) allows for strictly more valid pattern matches than (B) at the

[1] Throughout this discussion, we assume that the data types in question have more than one constructor. A data type with a single non-inductive constructor of arity k is equivalent to a k-tuple. Pattern matching here amounts to extracting its components which has the same efficiency problems as tuples, namely the trade-off between lazily or eagerly decomposing such data types. See Section 4.1.1 for ways of dealing with these problems.

cost of potentially dealing with synthesizing branches without examples. However, the performance gains with using restriction (B) over (A) are smaller than might be expected for a pair of reasons:

1. For many simple synthesis problems of the sort that we explore in Chapter 8, we deal with algebraic data types that only have two constructors—a base and inductive constructor. In this setting, restrictions (A) and (B) are isomorphic.

2. More generally, it turns out that almost all scrutinees fall into the extreme categories—completely uninformative or very informative, according to the definition of informativeness given above. That is, if a pattern match is actually informative, it will send examples to all of its branches as long as there is an appropriate number of examples to begin with.

Because of this, either restriction is suitable for recovering performance in the presence of matches. When referring to these restrictions in future discussion, we'll refer to a single "Informativeness Restriction" but really mean either Informativeness Restriction (A) or (B).

Repetitive Matches The informativeness restriction takes care of situations where pattern matches do not help us make progress. However, there are other problems with matches that we must consider. For example, our collection semantics does not keep us from pattern matching on the same expression more than once. We could solve this by tracking the expressions that we pattern match over, but then we are still susceptible to chains of *equivalent yet redundant scrutinees* such as

```
match l with
...
  match append l [] with
  ...
    match append [] l with
    ...
      match append (append [] l) [] with
      ....
```

This means that the problem of repetitive matches is really a problem of *program equivalence* where we want to avoid generating semantically equivalent terms as scrutinees.

Thankfully in this situation, our informative restriction are sufficient to rule out redundant matches as Lemma 7.3.1 shows:

Lemma 7.3.1 (Informativeness Rules Out Repetition). *Consider a pattern match over some scrutinee e of some data type with constructors $C_1, \ldots, C_m$ and an expression e' such*

that e is equivalent to e'. Then any inner pattern match whose scrutinee is e' in any of these branches is invalid with respect to our informativeness restriction.

Proof. distribute creates example worlds $X_1, \ldots, X_m$ for each branch of the pattern match by evaluating e. Consider a single such branch $k \in 1, \ldots, m$ and pattern matching on a scrutinee e' that is equivalent to e in that branch. (It could be e itself). By definition X_k contains only the example worlds under which e evaluated to a constructor value with head C_k. Therefore, all of the example worlds in X_k will evaluate to the same constructor value with head C_k. This means that the inner pattern match will send all of its examples to branch C_k which by definition of both Informativeness Restrictions is invalid. □

Thus, we do not need to do anything special to handle repetitive pattern matches as long as we employ the informativeness restriction to prune down the space of possible match expressions.

Restricting Generation of Matches The informativeness restriction that we have developed in this section is designed to limit pattern matches that don't make progress towards the synthesis goal or otherwise cause us to degenerate to raw-term enumeration. With either restriction in effect, it is clear that we can no longer generate match expressions in the absence of examples. Therefore, it no longer makes sense to try to generate pattern matches during E-term generation where we explicitly enumerate terms in the absence of examples.

In Figure 7.2, we took advantage of the fact that the **refine** function degenerated to raw term enumeration when given no examples. This occurs in one place where we need to generate an I-form in the absence of examples: generating a function argument in the **gen** function. However, if we avoid pattern match generation in this situation altogether, we require a separate generation function for introduction forms rather than relying on the degenerate behavior of **refine**.

Figure 7.4 splits up raw-term enumeration into two functions, $\mathbf{gen}_E$ and $\mathbf{gen}_I$, which enumerates E- and I-forms respectively. The refinement portions of our collection semantics in Figure 7.2 are modified to call either generation function when an E- or I-form must be generated without examples. Notably, we never generate match expressions in either raw-term generation function because of our informativeness restrictions. This implies that we will never have a match expression appear as an argument to a function or constructor. However, this does not change the expressiveness of our synthesis procedure as any pattern match that occurred within the arguments of an application or constructor can be hoisted outside of it. In our collection semantics, we obtain this result by generating a match via **match** first and then generating the application through **gen** or constructor through **refine**.

$$\boxed{\mathbf{gen}_E(\Sigma;\Gamma;\tau;k) = \mathcal{E}}$$

$$\begin{array}{l}
\mathbf{gen}_E(\Sigma;\Gamma;\tau;0) = \{\} \\
\mathbf{gen}_E(\Sigma;\Gamma;\tau;1) = \{x \mid x{:}\tau \in \Gamma\} \\
\mathbf{gen}_E(\Sigma;\Gamma;\tau;k) = \bigcup_{x:\tau_1\to\tau\in\Gamma} \bigcup_{\substack{k_1,k_2 \text{ for} \\ 1+k_1+k_2=k}} \\
\quad \{E\ I \mid E \in \mathbf{gen}_E(\Sigma;\Gamma;\tau_1\to\tau;k_1), I \in \mathbf{gen}_I(\Sigma;\Gamma;\tau_1;k_2), \mathbf{struct}(\Gamma,E,I)\}
\end{array}$$

$$\boxed{\mathbf{gen}_I(\Sigma;\Gamma;\tau;k) = \mathcal{I}}$$

$$\begin{array}{l}
\mathbf{gen}_I(\Sigma;\Gamma;\tau;0) = \{\} \\
\mathbf{gen}_I(\Sigma;\Gamma;\tau_1\to\tau_2;k) = \{\lambda x{:}\tau_1.\ I \mid I \in \mathbf{gen}_I(\Sigma;x{:}\tau_1,\Gamma;\tau_2;k-1)\} \\
\mathbf{gen}_I(\Sigma;\Gamma;T;k) = \mathbf{gen}_E(\Sigma;\Gamma;T;k) \cup \\
\quad \{C(I_1,\ldots,I_m) \mid C : \tau_1 * \ldots * t_m \to T \in \Sigma, \\
\qquad 1+k_1+\ldots+k_m = k, \overline{I_j \in \mathbf{gen}_I(\Sigma;\Gamma;\tau_j;k_j)}^{j<m}\}
\end{array}$$

Figure 7.4: ML_{syn} raw-term collection semantics

7.3.3 Refinement Trees

The synthesis procedure we described in Section 7.2 performs a ton of redundant work. A naïve implementation of our iterative deepening search strategy coupled with the decomposition of types and examples means that we end up solving many identical synthesis subproblems throughout the synthesis process. To mitigate this cost we pre-compute and cache the results of our various collection semantics functions whenever possible.

First, let's reexamine the synthesis tree structure we discussed in Section 7.1. As discussed in Section 7.2, we can divide up the rules of our synthesis judgment into refinement, guessing, and matching rules. Note that the refinement rules are dictated entirely by the goal type and examples. In particular, IREFINE-ARR applies at arrow type at any time and IREFINE-BASE applies at base type whenever the examples agree on their head constructor. Both rules always decomposes both the goal type and the examples on every invocation. Because our goal type and examples are necessarily finite structures, this leads to an important property of I-refinements with respect to the synthesis tree structure.

Lemma 7.3.2 (Finiteness of Refinement Derivations). *In any branch of a synthesis tree, the number of* IREFINE-ARR *and* IREFINE-BASE *applications is finite.*

Proof. We make the stronger claim that the number of such applications is bounded by the sum of the goal type and the size of the example values. Consider the effects of each IREFINE rule on the synthesis state.

- IREFINE-GUESS ends the branch with a series of EGUESS derivations. If a EGUESS derivation calls back into the IREFINE judgment, it is without any examples.
- IREFINE-ARR decomposes the goal type and example values while growing number of example worlds.
- IREFINE-BASE decomposes the example values, keeps the number of example worlds, and may grow the goal type, *e.g.*, moving from a base type to an arrow type if one of the constructor arguments is higher order.
- IREFINE-GUESS distributes the examples, keeping the size of the examples and the goal type the size, but decreasing the number of example worlds.

From this, it is clear that the number of possible IREFINE-ARR and IREFINE-BASE applications in a single branch is finite because the example values are necessarily finite and no IREFINE rule grows those example values. □

Now, let's consider applications of IREFINE-MATCH. There are two dimensions of match applications to consider:

Width: At any given point in the synthesis tree, if we have a way of generating an infinite number of expressions of base type, *e.g.*, lists and the append function, then we can use those expressions as scrutinees of an infinite number of match expressions.

Depth: In any given branch of a synthesis tree, we may be able to generate an infinite chain of nested matches.

Our informativeness restriction from Section 7.3.2 ensures that match depth is finite. In contrast, we must add additional restrictions to ensure that the number of possible scrutinees at any given point in a synthesis tree is finite.

With these considerations, we can formulate a lemma about the finiteness of IREFINE-MATCH applications in any synthesis tree:

Lemma 7.3.3 (Finiteness of Match Derivations). *Suppose that we fix the maximum size of any match scrutinee expression to some constant k. Then, in any synthesis tree, the number of possible* IREFINE-MATCH *applications is finite (assuming that we apply an informativeness restriction on match expressions).*

Proof. Our bound k on the size of match scrutinees artificially limits match width. As mentioned above, match depth is limited by our informativeness restriction. To see why, note that match expressions distribute the example worlds among the branches of the match. The number of example worlds does not grow.[2] Furthermore, our informativeness restriction ensures that once we reach branches with one or no examples, we will not be able to pattern match further as we will not be able to fulfill the requirement that at least two branches of the inner pattern match have examples.[3] □

Lemma 7.3.2 and Lemma 7.3.3 assert that type-directed refinement—application of the IREFINE rules—is finite given a particular synthesis problem. In contrast, it is clear that E-guessing or raw-term generation is potentially limitless depending on the input context Γ. In light of this revelation, we can separate I-refinement from E-guessing completely. Rather than searching the space of derivations given by a synthesis tree all at once, we can break up our synthesis procedure into two phases. Given a synthesis problem (a context Γ, a goal type τ, and examples X):

1. Compute the portion of the synthesis tree corresponding to the set of possible I-refinements for the synthesis problem.

2. Search the space of possible E-guesses that correspond to the remainder of the synthesis tree.

We call the portion of the synthesis tree pre-calculated in step (1) a *refinement tree* which describes the possible shapes of satisfying programs as dictated by the synthesis specification—the goal type and examples. For example, Figure 7.1 gives the synthesis tree corresponding to the synthesis problem we explored in Section 5.3 where our goal type was $\mathsf{list} \to \mathsf{list}$ and our examples were

$$\begin{aligned} &[] \Rightarrow [] \\ \mid\ &[0] \Rightarrow [0, 0] \\ \mid\ &[1, 0] \Rightarrow [1, 1, 0, 0] \end{aligned}$$

which correspond to the stutter function.

We can construct a refinement tree by slightly modifying our collection semantics from Section 7.2. Figure 7.6 gives the definition of **rtree** which creates a refinement tree given a signature Σ, context Γ, goal type τ, examples X, maximum

[2] From the proof of 7.3.2, we see that the number of example worlds increases when applying IREFINE-ARR. However, these additional example worlds are drawn from the structure of the goal example values, so it is really the size of the example values that is decreasing here.

[3] This requires that we handle pattern matches over data types with single constructors differently, *e.g.*, greedily expanding them with focusing as described in Section 4.1.1, as the informativeness restriction does not apply to them.

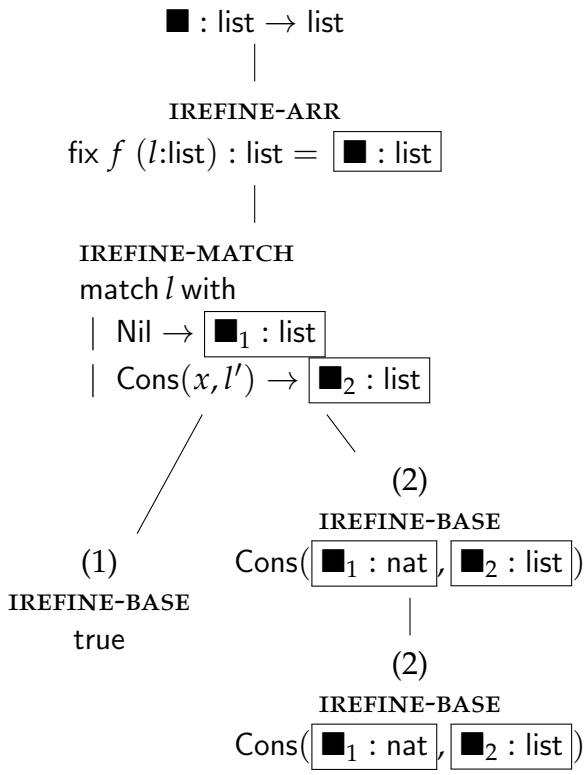


Figure 7.5: Example refinement tree.

$$\mathbf{types}(\Sigma) = \{T \mid C : \tau_1 * \ldots * \tau_m \to T \in \Sigma\}$$
$$\mathbf{rmatch}^s(\Sigma;\Gamma;\tau;X;0) = \{\}$$
$$\mathbf{rmatch}^s(\Sigma;\Gamma;\tau;X;k) = \bigcup_{T \in \mathbf{types}(\Sigma)} \{\underline{\mathsf{match}\ E\ \mathsf{with}\ \overline{p_i \to \blacksquare_i}^{i<m}} \mid$$
$$E \in \mathbf{gen}_E(\Sigma;\Gamma;T;s),$$
$$\overline{(p_i, X_i)}^{i<m} = \mathbf{distribute}(\Sigma, T, X, E),$$
$$\overline{\Gamma_i}^{i<m} = \mathbf{binders}(\Gamma, E, p_i),$$
$$\overline{\blacksquare_i = \mathbf{rtree}^s(\Sigma;\Gamma_i,\Gamma;\tau;X_i;k-1)}^{i<m}\}$$

$$\mathbf{rtree}^s(\Sigma;\Gamma;\tau_1 \to \tau_2;X;k) = \{\underline{\mathsf{fix}\ f\ (x{:}\tau_1) : \tau_2 = \blacksquare} \mid$$
$$X' = \mathbf{apply}(f, x, \sigma_1 \mapsto \rho_1) +\!\!+ \ldots +\!\!+ \mathbf{apply}(f, x, \sigma_n \mapsto \rho_n),$$
$$\blacksquare = \mathbf{rtree}^s(\Sigma; f{:}\tau_1 \to \tau_2\{\mathsf{rec}\}, x{:}\tau_1\{\mathsf{arg}\ f\};\Gamma;X';k)$$
$$\}$$
where
$$X = \sigma_1 \mapsto \rho_1, \ldots, \sigma_n \mapsto \rho_n$$

$$\mathbf{rtree}^s(\Sigma;\Gamma;T;X;k) = \{\underline{C(\blacksquare_1, \ldots, \blacksquare_m)} \mid$$
$$X_1, \ldots, X_m = \mathbf{proj}(X),$$
$$\overline{\blacksquare_i \in \mathbf{rtree}^s(\Sigma;\Gamma;\tau_i;X_i;k)}^{i<m}$$
$$\} \cup \mathbf{rmatch}^s(\Sigma;\Gamma;\tau;X;k)$$
where
$$X = \overline{\sigma_i \mapsto C(I_{i1}, \ldots, C_{im})}^{i<n}$$
$$C : \tau_1 * \ldots * \tau_m \in \Sigma$$

Figure 7.6: ML_{syn} refinement tree creation

number of matches k, and maximum scrutinee size s. **rtree** behaves similarly to **refine** except that rather than constructing complete programs of a certain size, **rtree** derives the set of possible I-refinements for a given synthesis problem without performing any E-guessing. To ensure finiteness (as per our discussion in Section 7.3.2), we must fix the scrutinee size of all matches to some fixed value, s. We also parameterize the number of matches that appear in any branch of the refinement tree by k. (Note that k no longer dictates overall program size.) This is unnecessary to ensure that the refinement tree is finite (*i.e.*, that **rtree** terminates), but is useful in a practical implementation as the branching induced by match expressions is still problematic for scaling. The **rtree** function produces a set of refinement tree nodes which carry the synthesis state as well as a possible I-term at that point in a synthesis derivation. We denote these with an underline to remind us that these are tree nodes rather than expressions, for example, $\underline{\mathsf{fix}\ f\ (x{:}\tau_1) : \tau_2 = \blacksquare}$ for a refinement tree node denoting a fix.

After using **rtree** to construct the refinement tree, we can now E-guess up to some fixed term size with the **guess** function defined in our original collection semantics (but calling into the raw-term enumeration functions we developed in Section 7.3.2 instead). We E-guess at every node in the refinement tree where the goal is a base type; these points are indicated by boxes in our example synthesis tree found in Figure 7.1. This these boxes coincide with invocations of the **guess** function in our original collection semantics.

Finally, if we are able synthesize a solution in some branch of the refinement tree either through refinement or guessing, we propagate the result up the tree to try to synthesize a solution to the original synthesis problem. For example, if we are able to E-guess a satisfying program E to the synthesis problem at a function node $\underline{\mathsf{fix}\ f\ (x{:}\tau_1) : \tau_2 = \blacksquare}$, then we are able to synthesize the satisfying program $\underline{\mathsf{fix}\ f\ (x{:}\tau_1) : \tau_2 = E}$ at that point in the refinement tree. We can then continue to propagate this function upwards in the refinement tree or return it as the result of the overall synthesis process if this node is the root of the tree.

Synthesis with Refinement Trees With refinement trees, we arrive at a much more efficient synthesis procedure observing that I-refinement and E-guessing are separable. I-refinements are dictated by the goal type and examples and so they can be computed first. From there, we can use raw-term enumeration to perform E-guessing as before. In summary, our new synthesis procedure with refinement trees operates as follows:

Given a signature Σ, context Γ, goal type τ, and examples X:

1. Let our *search parameters* be s, the maximum match scrutinee size, m, the maximum match depth, and k, the maximum E-term size.

2. Generate a refinement tree, $\mathbf{refine}^s(\Sigma; \Gamma; \tau; m)$.

3. Perform E-guessing (using **guess**) at each node of the refinement tree whose goal is a base type.

4. Propagate successfully synthesized problems upwards in the refinement tree to try to synthesize a solution to the overall synthesis problem.

5. If we are unable to create such a solution, increment our search parameters and repeat starting at step (2).

With this algorithm, we may not synthesize smallest program possible as we are no longer exploring the synthesis tree in a breadth-first manner. We can think of synthesis with refinement trees as performing some partial depth-first search to explore the set of refinements before proceeding in a breadth first manner with matches and E-guessing. The precise description of the programs generated by this algorithm is dependent on our choice of how we explore the search parameters on each iteration of the search. We discuss our particular choice of search metric exploration in Chapter 8. Suffice to say, in practice, the programs that this approach generates are usually identical to the smallest programs guaranteed by straightforward breadth-first search. If not, then they are either equivalent but larger or require an extra example or two to produce the desired program.

As an example, consider applying this procedure towards synthesizing the stutter function of type list $\to$ list using the examples:

$$\begin{array}{l} [] \Rightarrow [] \\ \mid [0] \Rightarrow [0,0] \\ \mid [1,0] \Rightarrow [1,1,0,0] \end{array}$$

from before. Let's suppose that we start with the following search parameters:

- Maximum match scrutinee size: $s = 1$.
- Maximum match depth: $m = 0$.
- Maximum E-term size: $k = 5$.

Then, the resulting refinement tree tree looks like this:

$$\blacksquare : \mathsf{list} \to \mathsf{list}$$

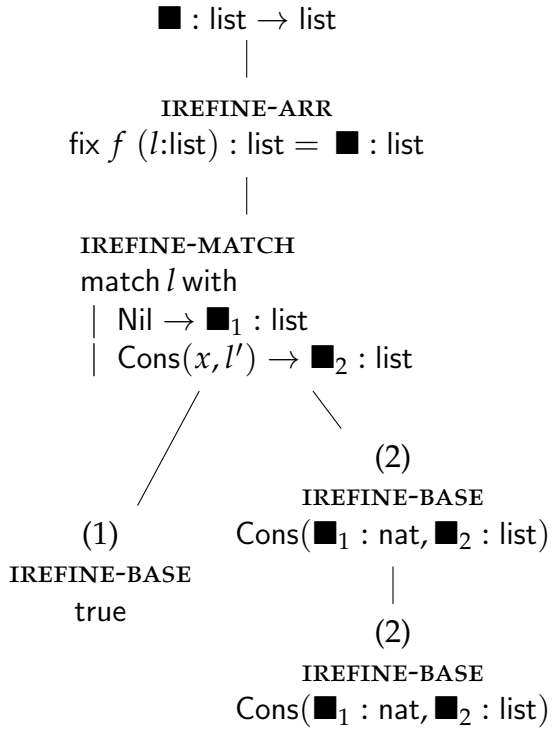
$$\Big|$$

IREFINE-ARR

$$\mathsf{fix}\ f\ (l{:}\mathsf{list}) : \mathsf{list} = \boxed{\blacksquare : \mathsf{list}}.$$

We can make no further progress because the examples at the IREFINE-ARR node do not share the same head constructor, and we are not allowed to introduce matches as the match depth is zero.

While we can E-guess terms up to size 5, the only term we can generate (in the empty context) is l which does not satisfy the body of the fix. We therefore need to iterate by incrementing our search parameters and repeating the process. Let's increase the match depth by one. The resulting refinement tree is:



which is precisely the refinement tree we presented in Figure 7.5. When performing E-guessing on this tree, we synthesize the following satisfying sub-expressions:

- x as the first argument to the first Cons constructor in the Cons branch of the pattern match,
- x as the first argument to the second Cons constructor in the Cons branch of the pattern match, and
- $f\ l'$ as the second argument to the second Cons constructor in the Cons branch of the pattern match.

Propagating these results upwards in the tree allows us to synthesize the final, expected stutter program:

$$
\begin{array}{l}
\mathsf{fix}\ f\ (l{:}\mathsf{list}) : \mathsf{list} = \\
\quad \mathsf{match}\ l\ \mathsf{with} \\
\quad \mid\ \mathsf{Nil} \rightarrow \mathsf{Nil} \\
\quad \mid\ \mathsf{Cons}(x, l') \rightarrow \mathsf{Cons}(x, \mathsf{Cons}(x, f\ l')).
\end{array}
$$

The biggest win we receive in factoring out E-guessing from I-refinement is that we have to enumerate far smaller terms. Rather than enumerating all terms of up to size 11 to derive stutter, we only need to enumerate terms up to size three to obtain the E-guessed expressions above. This is a dramatic savings in term enumeration which turns out to be the bottleneck in our type-directed synthesis style.

Furthermore, this factoring allows us to take advantage of the fact that we decompose our synthesis problems into smaller, independent synthesis sub-problems. For example, in the second iteration of our search, we do not need to E-guess terms in the Nil branch of the pattern match because we already have a completed expression for that branch, Nil, which we obtained in the refinement tree portion of the synthesis process. After discovering that x is a valid completion as the first arguments of the Cons constructors, we do not need to enumerate larger terms at those points in the refinement tree because any valid solution there is equivalent to x with respect to satisfying the examples. This ability to *short-circuit* synthesis once we find local solutions to synthesis problems is instrumental to scaling up our synthesis procedure. This means that if one branch of a pattern match requires a large expression, but the remaining branches require small expressions, we only pay the (exponential) cost of searching for the large expression in that one branch, rather than among all of the branches by performing unnecessary search.

7.3.4 Efficient Raw-term Enumeration

Our refinement tree structure allows us to cache I-refinements, allowing us to optimize the synthesis procedure greatly. Can we perform similar caching for E-guessing as well? We observe that the inefficiency here lies in repetitive calls to our term-generation functions $\mathbf{gen}_E$ and $\mathbf{gen}_I$. For example, we may need to repeatedly call $\mathbf{gen}_E(\Sigma; \Gamma; \mathsf{list}; k)$ to synthesize expressions of type list as the potential body of our function, the leaf expression of a pattern match branch, as an argument to a function application, or otherwise part of some complex expression we are building up. Ideally, we should cache the results of $\mathbf{gen}_E$ and $\mathbf{gen}_I$ for particular combinations of arguments to ensure that we only ever generate a particular term once during the enumeration process.

If we examine the arguments to **gen** we see that the signature Σ remains constant and the goal type τ and term size k are natural "keys" by which we can cache results to **gen**. However, our context Γ does not remain constant throughout the synthesis process. For example, in our refinement tree example above, we generate terms of type list in several contexts:

- $\Gamma_1 = f{:}\mathsf{list} \to \mathsf{list}, l{:}\mathsf{list}$ just inside the body of the fix and in the Nil branch of the match.
- $\Gamma_2 = x{:}\mathsf{nat}, l'{:}\mathsf{list}, f{:}\mathsf{list} \to \mathsf{list}, l{:}\mathsf{list}$ in the Cons branch of the match.

$$\boxed{\mathbf{gen}_E(\Sigma;\Gamma;\tau;n)}$$

$$\begin{aligned}
\mathbf{gen}_E(\Sigma;\cdot;\tau;n) &= \{\} \\
\mathbf{gen}_E(\Sigma;\cdot;\tau;0) &= \{\} \\
\mathbf{gen}_E(\Sigma;x{:}\tau_1,\Gamma;\tau;n) &= \mathbf{gen}_E^{x:\tau_1}(\Sigma;\Gamma;\tau;n) \cup \mathbf{gen}_E(\Sigma;\Gamma;\tau;n)
\end{aligned}$$

$$\boxed{\mathbf{gen}_E^{x:\tau_1}(\Sigma;\Gamma;\tau;n)}$$

$$\begin{aligned}
\mathbf{gen}_E^{x:\tau_1}(\Sigma;\Gamma;\tau;0) &= \{\} \\
\mathbf{gen}_E^{x:\tau}(\Sigma;\Gamma;\tau;1) &= \{x\} \\
\mathbf{gen}_E^{x:\tau_1}(\Sigma;\Gamma;\tau;1) &= \{\} \qquad (\tau \neq \tau_1) \\
\mathbf{gen}_E^{x:\tau_1}(\Sigma;\Gamma;\tau;n) &= \bigcup_{\tau_2\to\tau\in\Gamma} \bigcup_{k=1}^{n-1} \\
&\quad (\mathbf{gen}_E^{x:\tau_1}(\Sigma;\Gamma;\tau_2\to\tau;k) \underset{app}{\otimes} \mathbf{gen}_I(\Sigma;\Gamma;\tau_2;n-k)) \\
&\cup\ (\mathbf{gen}_E(\Sigma;\Gamma;\tau_2\to\tau;k) \underset{app}{\otimes} \mathbf{gen}_I^{x:\tau_1}(\Sigma;\Gamma;\tau_2;n-k)) \\
&\cup\ (\mathbf{gen}_E^{x:\tau_1}(\Sigma;\Gamma;\tau_2\to\tau;k) \underset{app}{\otimes} \mathbf{gen}_I^{x:\tau_1}(\Sigma;\Gamma;\tau_2;n-k))
\end{aligned}$$

Figure 7.7: Relevant E-term generation

Clearly, we cannot interchange the results of $\mathbf{gen}_E(\Sigma;\Gamma_1;\mathsf{list};k)$ and $\mathbf{gen}_E(\Sigma;\Gamma_2;\mathsf{list};k)$ because they contain different sets of expressions. But the two calls to **gen** also clearly share some expressions in common. We would like to be be able to realize this sharing in our term caches as well to avoid redundant work.

To do this, we present a technique for efficiently performing term enumeration in the presence of contexts called *relevant term generation*. Critically, we note that our contexts during the synthesis process only grow; they never shrink or shuffle their contents. As a result, we can factor the term generation function as follows:

$$\begin{aligned}
\mathbf{gen}_E(\Sigma;x{:}\tau_1,\Gamma;\tau;n) &= \mathbf{gen}_E^{x:\tau_1}(\Sigma;\Gamma;\tau;n) \\
&\quad \cup \mathbf{gen}_E(\Sigma;\Gamma;\tau;n) \\
\mathbf{gen}_I(\Sigma;x{:}\tau_1,\Gamma;\tau;n) &= \mathbf{gen}_I^{x:\tau_1}(\Sigma;\Gamma;\tau;n) \\
&\quad \cup \mathbf{gen}_I(\Sigma;\Gamma;\tau;n)
\end{aligned}$$

This factorization ensures that for a given goal type τ and size n, two calls to **gen** in different contexts Γ and $x{:}\tau_1,\Gamma$ share the same set terms under the shared context Γ.

Here, $\mathbf{gen}_E^{x:\tau_1}$ and $\mathbf{gen}_I^{x:\tau_1}$ are *relevant term generation functions*. Inspired by

$$\boxed{\mathbf{gen}_I(\Sigma, \Gamma, \tau, n)}$$

$$\begin{aligned}
\mathbf{gen}_I(\Sigma; \cdot; \tau; n) &= \{\} \\
\mathbf{gen}_I(\Sigma; \Gamma; \tau; 0) &= \{\} \\
\mathbf{gen}_I(\Sigma; x : \tau_1, \Gamma; \tau; n) &= \mathbf{gen}_I^{x:\tau_1}(\Sigma; \Gamma; \tau; n) \cup \mathbf{gen}_I(\Sigma; \Gamma; \tau; n) \\
\mathbf{gen}_I(\Sigma; \cdot; \tau_1 \to \tau_2; n) &= \\
&\{\mathsf{fix}\ f\ (x{:}\tau_1) : \tau_2 = I \mid I \in \mathbf{gen}_I(\Sigma; f : \tau_1 \to \tau_2, x : \tau_1; \tau_2; n-1)\} \\
\mathbf{gen}_I^{x:\tau}(\Sigma; \Gamma; T; n) &= \mathbf{gen}_E^{x:\tau}(\Sigma; \Gamma; T; n) \\
&\bigcup_{C:\tau_1 * \ldots * \tau_k \to T \in \Sigma} \ \bigcup_{\substack{n_1,\ldots,n_k \text{ for} \\ n_1+\ldots+n_k=n}} \\
&\{C(I_1, \ldots, I_k) \mid I_j \in \mathbf{gen}_I(\Sigma; \Gamma; \tau_j; n_j)\}
\end{aligned}$$

$$\boxed{\mathbf{gen}_I^{x:\tau_1}(\Sigma; \Gamma; \tau; n)}$$

$$\begin{aligned}
\mathbf{gen}_I^{x:\tau}(\Sigma; \Gamma; \tau_1 \to \tau_2; n) &= \mathbf{gen}_E^{x:\tau}(\Sigma; \Gamma; \tau_1 \to \tau_2; n) \cup \\
&\{\mathsf{fix}\ f\ (y{:}\tau_1) : \tau_2 = I \mid I \in \mathbf{gen}_I^{x:\tau}(\Sigma; f : \tau_1 \to \tau_2, y : \tau_1, \Gamma; \tau_2; n-1)\}
\end{aligned}$$

$$\begin{aligned}
\mathbf{gen}_I^{x:\tau}(\Sigma; \Gamma; T; n) &= \mathbf{gen}_E^{x:\tau}(\Sigma; \Gamma; T; n) \\
&\bigcup_{C:\tau_1 * \ldots * \tau_k \to T \in \Sigma} \ \bigcup_{\substack{n_1,\ldots,n_k \text{ for} \\ n_1+\ldots+n_k=n}} \ \bigcup_{\substack{r_1,\ldots,r_k \in \\ \mathbf{parts}(k)}} \\
&\{C(I_1, \ldots, I_k) \mid I_j \in \mathbf{genp}_I^{m_j;x:\tau}(\Sigma; \Gamma; \tau_j; n_j)\}
\end{aligned}$$

$$\mathbf{parts}(k) = \{\underbrace{\mathsf{Not}, \ldots, \mathsf{Not}}_{i-1}, \mathsf{Must}, \underbrace{\mathsf{May}, \ldots, \mathsf{May}}_{k-i} \mid i \in 1, \ldots, k\}$$

$$\mathbf{genp}_I^{r;x:\tau_1}(\Sigma; \Gamma; \tau; n) = \begin{cases} \mathbf{gen}_I^{x:\tau_1}(\Sigma; \Gamma; \tau; n) & r = \mathsf{Must} \\ \mathbf{gen}_I(\Sigma; x : \tau_1, \Gamma; \tau; n) & r = \mathsf{May} \\ \mathbf{gen}_I(\Sigma; \Gamma; \tau; n) & r = \mathsf{Not} \end{cases}$$

Figure 7.8: Relevant I-term generation

relevance logic [Anderson et al., 1992], these functions are variants of our standard term-enumeration functions except that they require that all expressions they generate must contain the relevant variable x. Figure 7.7 and Figure 7.8 gives the definition of relevant E- and I-term generation. The functions operate similarly

to the **gen** functions we developed in Figure 7.4. The critical difference is that when our relevant E-term generation functions bottoms out at size one, rather than generating all terms of goal type τ, we generate only a single term, the relevant variable x, when the goal type is the relevant variable's type.

The relevant term-generation functions ensure that the relevant variable x appears in every term generated by the function. When we generate terms that contain multiple sub-expressions in a relevant context, we must be careful to ensure that this property holds. For example, we can break up generation of a function application $E\ I$ with a relevant variable x into three cases:

1. x must appear in E and must not appear in I.
2. x must not appear in E and must appear in I.
3. x must appear in both E and I.

These cases are reflected in the definition of relevant term generation for function applications in Figure 7.7. To ensure that x appears in a particular sub-term, we invoke the relevant term generation function with x, $\mathbf{gen}_E^{x:\tau}$ or $\mathbf{gen}_I^{x:\tau}$. To ensure that x does not appear in a particular sub-term, we invoke the non-relevant term generation function in a context not containing x.

For constructors (Figure 7.8), we must generalize this factorization to k subexpressions rather than just two. To do this, employ a "sliding window" factorization (realized by the parts helper function) where we walk the list of sub-expressions, distinguishing the current expression I_m as the one that *must* contain the relevant variable x. Throughout this process, we note that x has been required to appear in all of the expressions before I_m. Therefore, we require that x *must not appear* in the expressions before I_m. In contrast, x *may* appear in the expressions after I_m as we have not placed any restrictions on them yet.

Now we have three cases of sub-term generation that we handle with the genp helper function. The cases where x must and must not appear in the sub-term are handled similarly to the function application case. To generate terms that may contain x, we appeal to the non-relevant term generation function, adding x into the context.

Chapter 8

Evaluating Myth

In Chapter 7, we developed an efficient synthesis procedure from our core synthesis calculus ML_{syn}. In this chapter, we explore our implementation of this synthesis procedure, a prototype program synthesizer called Myth.[1]

Our goal with Myth is to further explore the type-theoretic foundations for program synthesis that we have developed so far. We started our exploration by carefully analyzing the metatheory of type-directed program synthesis, in particular the soundness and completeness of $\lambda^{\rightarrow}_{syn}$ and ML_{syn}. However, this is insufficient for getting a complete sense of how program synthesis systems built on top of these foundations will perform in practice. By exploring the behavior of an actual implementation, we can better understand the capabilities and limitations of our approach and identify areas for future improvement.

Note that an explicit non-goal of this exploration is to justify Myth-the-artifact as a practical tool for program synthesis. While we explore some aspects of the viability of Myth as an end-user tool, *e.g.*, performance, we intentionally do not explore the usability of the tool. We do this primarily as a matter of pragmatics. There are plenty of empirical questions to investigate about Myth—How many examples do we need to synthesize a particular program? How long does it take to synthesize a particular program?—without delving into the usability side of the project. However, we also want to stress that, throughout this work, we have been less concerned with building a practical tool and more interested in answering foundational questions about the integration of types into program synthesis. We do not want to overshadow these important results with claims about usability that we do not have the time to develop thoroughly. We leave such investigation to future work.

[1] We provide a summary of the final synthesis procedure as a reference in Appendix A.

8.1 Search Parameter Tuning

Originally we started with a simple synthesis procedure that searched the space of programs according to program size in a breadth-first manner. When we introduced refinement trees, we also introduced a number of search parameters into the procedure:

- s, the maximum scrutinee size of any match expression,
- m, the maximum match depth, *i.e.*, the maximum number of matches that can appear in any branch of a refinement tree, and
- k, the maximum size of E-terms that we generate during the E-guess phase of the procedure.

Rather than performing a breadth-first search by program size alone, we now perform a breadth-first search according to these three parameters. Thus, choosing appropriate *initial values* along with a *strategy* for traversing through successive iterations of the algorithm is imperative for obtaining good synthesis results.

Scrutinee Size Let's consider the effects of each of these parameters on the programs that the synthesis algorithm produces. First, the scrutinee size affects the complexity of the pattern matches that we can synthesize. At $s = 1$, we can only synthesize variables as scrutinees. This has been sufficient for the examples we have used so far, but in general we need the ability to synthesize richer scrutinees. As an example, suppose in OCaml that we have the standard lookup function of type $'a \to ('a, 'b)\ \mathsf{list} \to' b\ \mathsf{option}$ in our context. Then we need richer scrutinees to synthesize programs that use association lists such as:

$$\begin{array}{l}\mathsf{match}\ (\mathsf{lookup}\ x\ l)\ \mathsf{with}\\ \mid\ \mathsf{Some}\ r \to \blacksquare\\ \mid\ \mathsf{None} \to \blacksquare\end{array}$$

However, we want to avoid opening the door to complex scrutinees too quickly. This is because the number of possible scrutinees (and consequently, the number of possible matches) grows exponentially with the scrutinee size and in the common case (based on personal experience and analysis of our benchmark suite), we only need to pattern match over a single variable. Our informativeness restriction rules out some of these scrutinees, but in practice, many scrutinees make it through the restriction, requiring us to add these branches to our refinement tree. These additional branches become more points where we have to E-guess, making synthesis more costly. Furthermore, on top of these scrutinees, as we increase the scrutinee size s, we also increase the likelihood of generating scrutinees that are equivalent to previously generated scrutinees.

Match Depth Related to the scrutinee size, the match depth m controls the number of match expressions that can appear in any branch of the refinement tree. Whereas the scrutinee size controls the *width* of matches, *i.e.*, the amount of possible match expressions at any point in a refinement tree, the match depth controls the *depth* of matches, *i.e.*, how deeply we can nest pattern matches in a candidate program. In many cases, a single match expression is sufficient to satisfy a given synthesis problem. However, sometimes we may also need to resort to nested matching to access successive elements of a structure or to make decisions based off of case analysis of multiple pieces of data. The danger is that unnecessary nested matches greatly impact performance because in addition to the branching factor, each branch introduces additional binders which accelerates the exponential blow-up associated with raw-term enumeration. Thankfully, we rule out equivalent, nested matches thanks to our informativeness restriction (see Section 7.3.2), although the cost of stacking up binders due to nested matches is very significant.

An additional problem with setting the match depth too high is overspecialization. For example, consider the overspecializing function for stutter from Section 7.2.1.

$$
\begin{array}{l}
\mathsf{fix}\ f\ (l{:}\mathsf{list}) : \mathsf{list} = \\
\quad \mathsf{match}\ l\ \mathsf{with} \\
\quad \mid\ \mathsf{Nil} \rightarrow [] \\
\quad \mid\ \mathsf{Cons}(x, l') \rightarrow \mathsf{match}\ l'\ \mathsf{with} \\
\quad\quad \mid\ \mathsf{Nil} \rightarrow [0, 0] \\
\quad\quad \mid\ \mathsf{Cons}(y, l'') \rightarrow \mathsf{match}\ l''\ \mathsf{with} \\
\quad\quad\quad \mid\ \mathsf{Nil} \rightarrow [1, 1, 0, 0] \\
\quad\quad\quad \mid\ \mathsf{Cons}(z, l''') \rightarrow [].
\end{array}
$$

By synthesizing programs in order of increasing size, we guaranteed that our original synthesis procedure would not synthesize this program before synthesizing the desired recursive function. However, imagine in our new synthesis procedure that we started with the match depth $m = 3$. Notice that because the overspecialization contains no E-guessed terms, our initial refinement tree would contain this program immediately and we would be done! Because of this, we have to start with a small match depth and be careful about increasing it too quickly.

***E*-term Size** Finally, k controls the maximum size of E-terms that the algorithm is allowed to guess at any node in the refinement tree. The primary benefit of our refinement tree structure, is that we have localized the expensive procedure of E-term enumeration to the leaves of the refinement tree. This allows us to generate much smaller E-terms on average. For example, in our original algorithm, when we synthesized terms up to size 11, we would need to explore the space of E-terms of size 11 if our context allowed us to generate E-terms of that size. In our updated algorithm, rather than having to synthesize E-terms up to size 11 to find stutter,

we only need to synthesize E-terms up to size 3 to find the recursive function call $f\ l'$. This is a tremendous win in terms of efficiency because of the exponential cost of enumerating large E-terms.

Unlike match depth, we do not run into issues of overspecialization if k begins too large. In fact, by admitting larger E-terms, we use more variables and function applications which, by our Minimum Programming Principle (Section 7.2.1) makes it more likely that our synthesized program generalizes to the behavior that the user intends. Furthermore, the cost of synthesizing E-terms of modest size is very small. It is not until we reach the critical point in the exponential cliff that generation time jumps from milliseconds to minutes. So we are safe in starting with an E-term size of reasonable size.

8.1.1 Search Strategy

With these considerations in mind, we can now discuss the particular search strategy we implemented in MYTH. With multiple search parameters, we could devise an *adaptive* strategy that responds to the performance of the synthesizer for the particular synthesize problem at hand. For example, we may vary the initial E-term size k or how much we increment the k depending on the size of the context we synthesize under. Because our context grows as we move down in the refinement tree as we add binders with functions or matches, we would need to vary the search parameters at different points in the tree.

For these experiments, we opted for a simpler *static* strategy suitable for the domain of programs we intended to synthesize—simple functional programs over recursive data types such as lists and trees. We begin with the following initial search parameter values:

- The maximum scrutinee size, $s = 1$.
- The maximum match depth, $m = 0$.
- The maximum E-guess size, $k = 13$.

We then move through the following phases of I-refinement. In each phase of I-refinement, we increase one of our search parameters and then extend the refinement tree accordingly. In particular, when we increase the match depth or scrutinee size, we traverse the refinement tree, create new match expressions as necessary, and carry out further I-refinements (in particular, application of IREFINE-ARR and IREFINE-BASE) where ever possible. Finally, after each I-refinement phase, we allocate 0.25 seconds to generates E-terms up to the size limit k at each eligible refinement tree node and then propagate the results to see if we can synthesize a complete, satisfying program.

The phases of I-refinement we go through are:

1. Create an initial refinement tree with the search parameters s and m.

```
type nat =
| O
| S of nat

type list =
| Nil
| Cons of nat * list

let list_stutter : list -> list |>
{ [] => []
| [0] => [0;0]
| [1;0] => [1;1;0;0]
} = ?
```

Figure 8.1: Example Myth program: stutter. (Throughout this chapter we typeset Myth source code and output in a fixed-width, syntax highlighted font.)

2. Increase m by one ($m = 1$) and extend the refinement tree accordingly.
3. Increase m by one ($m = 2$) and extend the refinement tree accordingly.
4. Increase s by five ($s = 6$) and extend the refinement tree accordingly.
5. Increase m by one ($m = 3$).

By the end of the process, we synthesize programs with E-terms of size 13, triply-nested pattern matches, with scrutinees up to size six. This is simple strategy is sufficient to handle all of the examples that we discuss in Section 8.3.

8.2 Example Development

The input to the Myth synthesizer is a series of top-level declarations in the subset of OCaml identified by ML_{syn} along with a synthesis problem—a name, goal type, and example values. For example, Figure 8.1 gives the source code for the stutter in Myth that we have studied extensively in the previous chapters. Myth has no built-in data types, so we must provide them all ourselves using algebraic data types. Here, we declare a type of monomorphic lists whose carrier type is nat. The declaration of stutter contains the familiar three input/output examples[2] suggesting the intended behavior of sutter.

We arrived at this set of examples by using the following process:

[2]Myth provides syntactic sugar for representing lists and natural numbers.

1. We observed that the likely data type we need to perform induction over was list, so we provided examples of how the function should work in both cases of list as defined its constructors: $[] \Rightarrow []$ | $[0] \Rightarrow [0, 0]$.

2. We ran Myth with these examples, obtaining the program:

```
let list_stutter : list -> list =
  fun (l1:list) -> match l1 with
                   | Nil -> []
                   | Cons (n1, l2) -> [0; 0]
;;
```

   Upon inspection, this program is not sufficient because while it has the correct behavior in the Nil case, it does not have the correct behavior in the Cons case, returning a constant list rather than performing a more general calculation.

3. To fix this problem, we need to add additional examples to rule out this expression in the Cons branch. Keeping in mind our trace completeness restriction to power recursive function calls (Section 5.3.4), we add one additional example that is trace complete with respect to the original examples, $[1, 0] \Rightarrow [1, 1, 0, 0]$, and see if that creates a satisfactory program. Updating our program with this additional example and running it through Myth yields:

```
let list_stutter : list -> list =
  let rec f1 (l1:list) : list =
    match l1 with
      | Nil -> []
      | Cons (n1, l2) -> Cons (n1, Cons (n1, f1 l2))
  in
    f1
;;
```

   Which is the standard implementation of stutter, so we are done!

We use this process of iterative example refinement based on the inductive structure of the data type to develop all of the programs we supply to Myth.

Multi-argument Functions When synthesizing single argument functions, we can perform case analysis using our examples as described above in a straight forward manner. However, the process becomes more complex with multi-argument

functions. In general, we will need additional examples demonstrating how our function behaves with each argument.

As a simple example, consider synthesizing the append function of type list $\rightarrow$ list $\rightarrow$ list. Since the only argument type is list, it makes sense that we ought to perform case analysis on one of the list arguments. Let's choose the first argument and try the following examples:

$$\begin{array}{l} \quad [] \Rightarrow [] \Rightarrow [] \\ \mid [0] \Rightarrow [] \Rightarrow [0] \\ \mid [1,0] \Rightarrow [] \Rightarrow [1,0] \end{array}$$

Feeding these examples to Myth produces the following function

```
let list_append : list -> list -> list =
  fun (l1:list) -> fun (l2:list) -> l1
;;
```

This is not the append function we want; it is the function that always chooses its first argument! Of course, looking at the examples we provided, this is certainly the simplest program that satisfies the examples.

The problem is that we have not specified the behavior of append on non-trivial second arguments. Let's try doing so incrementally, first introducing a single extra example:

$$\begin{array}{l} \quad [] \Rightarrow ([] \Rightarrow [] \ \mid \ [0] \Rightarrow [0]) \\ \mid [0] \Rightarrow [] \Rightarrow [0] \\ \mid [1,0] \Rightarrow [] \Rightarrow [1,0] \end{array}$$

Running Myth on these examples yields the refined function:

```
let list_append : list -> list -> list =
  fun (l1:list) ->
    fun (l2:list) -> match l2 with
                     | Nil -> l1
                     | Cons (n1, l3) -> Cons (0, l1)
;;
```

This function is closer to append but still not there. In particular, it only appends a 0 onto l_1 in the Cons case, completely ignoring l_2 in the process. Adding another example to handle when we call append $[0]$ $[0]$ results in the same program. We must add one additional example,

$$\begin{array}{l} \quad [] \Rightarrow ([] \Rightarrow [] \ \mid \ [0] \Rightarrow [0]) \\ \mid [0] \Rightarrow ([] \Rightarrow [0] \ \mid \ [0] \Rightarrow [0,0]) \\ \mid [1,0] \Rightarrow ([] \Rightarrow [1,0] \ \mid \ [0] \Rightarrow [1,0,0]). \end{array}$$

On these examples Myth produces the program

```
let list_append : list -> list -> list =
  let rec f1 (l1:list) : list -> list =
    fun (l2:list) ->
      match l1 with
        | Nil -> l2
        | Cons (n1, l3) -> Cons (n1, f1 l3 l2)
  in
    f1
;;
```

This final program is the standard implementation of append.

Validation An immediate question one might ask about this process is: "How do you validate that the programs that Myth produces are correct?" Indeed, this proves to be a tricky issue. Even though we proved soundness of the system (Lemma 6.2.1 and Lemma 6.2.2), this only guarantees that the programs Myth produces are well-typed and satisfy the examples. However, concrete examples only specify a finite subset of behavior which is insufficient to express the behavior of most functions we care to synthesis such as recursive functions. We appeal to the Minimum Program Principle (Section 7.2.1) to maximize the likelihood that the synthesized program agrees with the behavior that the user intends with their examples. However, at the end of the day, the user needs to validate for themselves that the resulting program meets their needs.

They might accomplish this by inspecting their program for obvious defects as we did in the above example. Or they may test their program on a wider variety of examples than what they presented to Myth. To validate our own examples, we went through this process. Because we synthesized common recursive functional programs, we were able to verify the correctness of Myth's output by inspection. In a minority of cases (some of which we explore in Section 8.4), the synthesized program's correctness was non-obvious and required additional testing to verify.

This work flow, as is, has some clear downsides in the context of an end-user tool. In particular, validation is problematic because in a real-world scenario, the user may not know what program they want in the end. After all, they are using program synthesis to try to discover this program! However, this work flow is adequate for discovering how our type-directed synthesis algorithm works and its behavior on real world examples.

8.3 Benchmark Suite

To assess the effectiveness of Myth, we created a synthesis benchmark suite consisting of 43 sample Myth programs developed using the methodology discussed

in Section 8.2. These benchmarks were developed in tandem with Myth to test the implementation as well as well as its expressiveness as we added features and optimized its synthesis engine. These programs provide specifications of simple functions over fundamental functional data types—booleans, natural numbers, lists, and trees—exercising all of the core functional programming features that Myth provides: inductive algebraic data types, higher-order functions, and recursion.

Booleans: Our simplest benchmarks involve synthesizing functions over booleans values. These programs include common boolean operators: neg, and, or, impl, and xor. Because these boolean functions are all non-recursive, we can fully specify their behavior through straightforward case analysis with our examples, demonstrating the power of pure type-directed refinement.

Natural Numbers: To graduate from the world from non-recursive functions to recursive functions, we introduce example programs over the most basic of inductive algebraic data types, the natural numbers:

```
type nat = O | S of nat.
```

These functions include unary functions over nats (*e.g.*, is_even and prev) as well as binary operators over nat (*e.g.*, max and sum).

Lists: Expanding on nats, the bulk of our benchmarks feature operations over lists. ML_{syn} and Myth do not have polymorphism. Therefore, our list data type is *monomorphic*, fixed to a particular carrier type, usually nat or bool. Our example programs include simple operations over a single list or pair of lists (*e.g.*, length and append) as well as more complex operations (*e.g.*, compress and reverse). Note that because our type-directed synthesis style does not not admit synthesis of let-bindings (Section 4.4), we must provide the appropriate helper functions to synthesize these programs. In particular, we explore synthesizing reverse with a variety of approaches: "cons-on-end" (snoc), append, fold, and a tail-recursive variant that requires a second argument. Finally, we use lists to explore synthesis of higher-order functions, providing specifications for implementations of map, fold and filter as well as usages of these higher-order functions as helpers.

Trees: Finally, we also explore richer data types with functions over tree data types which require richer sets of examples to capture their behavior. These operations include simple tree processing functions such as counting the number of nodes and performing different sorts of traversals over a tree as well as more complicated operations such as binary insertion (assuming that the tree is a binary search tree).

The full text of all these sample Myth programs as well as the resulting synthesized programs can be found in Appendix B.

8.3.1 Analysis

Now, let us examine the results of running Myth over our benchmark suite. In this section, we present performance numbers for an implementation of Myth in the OCaml programming language [Leroy et al., 2014]. This implementation was exercised on a desktop PC equipped with the openSUSE operating system (version 13.1), an Intel i7-3770 quad-core CPU clocked at 3.40 GHz, and 8 Gb of ram. Note that because the version of OCaml we used features a runtime with a global lock, we were unable to take advantage of more than one core. We expect that there are substantial performance benefits to moving to an implementation that can utilize multiple cores. All benchmarks were run five times in succession, and we report the average runtime collected from those trials.

Figure 8.2 presents the complete performance data of running Myth over our benchmark suite. The benchmark suite contains 42 test programs. On average, we provided Myth with 7 examples and it produced a program of size 13 in 0.092 seconds. Since every program that we synthesize is a function, we count an example as a single collection of input/output examples that determines one execution of the function in question. For example, we count the partial function we provided to Myth to synthesis the append function,

$$\begin{aligned} &[] \Rightarrow ([] \Rightarrow [] \ \mid \ [0] \Rightarrow [0]) \\ \mid\ &[0] \Rightarrow ([] \Rightarrow [0] \ \mid \ [0] \Rightarrow [0,0]) \\ \mid\ &[1,0] \Rightarrow ([] \Rightarrow [1,0] \ \mid \ [0] \Rightarrow [1,0,0]), \end{aligned}$$

as six examples, corresponding to the following executions of append:

$$\begin{aligned} \mathsf{append}\ []\ []\ &= [] \\ \mathsf{append}\ []\ [0]\ &= [0] \\ \mathsf{append}\ [0]\ []\ &= [0] \\ \mathsf{append}\ [0]\ [0]\ &= [0,0] \\ \mathsf{append}\ [1,0]\ []\ &= [1,0] \\ \mathsf{append}\ [1,0]\ [0]\ &= [0,0,0] \end{aligned}$$

The number of examples given to Myth for these benchmarks were the *minimal amount*, *i.e.*, removing any examples from that set would result in Myth synthesizing an incorrect program.

Examples and Program Complexity Let's take a deeper look at the data, first by examining the interplay between the number of examples required to synthesize a program and its final size. From Section 8.2, we saw that examples in Myth have the practical effect of forcing Myth to discriminate between cases. For example,

Test	#/Examples	Prog. size	Time (s)
	Booleans		
bool_band	4	6	0.0028
bool_bor	4	6	0.0024
bool_impl	4	6	0.0024
bool_neg	2	5	0.001
bool_xor	4	9	0.0026
	Lists		
list_append	6	12	0.005
list_compress	13	28	0.0932
list_concat	6	11	0.008
list_drop	11	13	0.013
list_even_parity	7	13	0.005
list_filter	8	15	0.016
list_fold	9	13	0.1724
list_hd	3	5	0.001
list_inc	4	8	0.001
list_last	6	11	0.003
list_length	3	8	0.001
list_map	8	12	0.0108
list_nth	13	16	0.0152
list_pairwise_swap	7	19	0.007
list_rev_append	5	13	0.0144
list_rev_fold	5	12	0.008
list_rev_snoc	5	11	0.0068
list_rev_tailcall	8	12	0.005
list_snoc	8	14	0.0036
list_sort_sorted_insert	7	11	0.0094
list_sorted_insert	12	24	0.1592
list_stutter	3	11	0.001
list_sum	3	8	0.0028
list_take	12	15	0.1488
list_tl	3	5	0.001
	Natural Numbers		
nat_iseven	4	10	0.001
nat_max	9	14	0.0166
nat_pred	3	5	0.001
nat_add	9	11	0.0028
	Trees		
tree_binsert	20	31	0.4834
tree_collect_leaves	6	15	0.0208
tree_count_leaves	7	14	0.0118
tree_count_nodes	6	14	0.0112
tree_inorder	5	15	0.0192
tree_map	7	15	0.017
tree_nodes_at_level	11	22	1.1078
tree_postorder	9	32	1.467

Figure 8.2: MYTH benchmark suite performance results

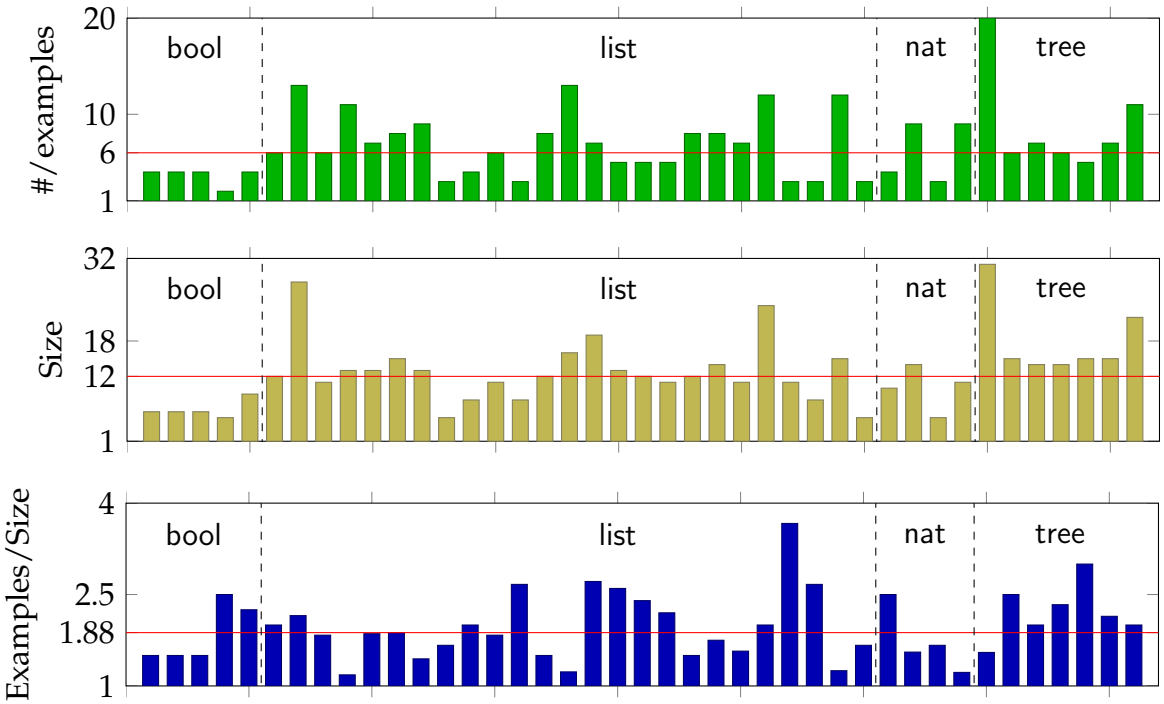


Figure 8.3: MYTH benchmark suite graphs—counts. Programs are grouped by the class of data type that they operate over. The first graph shows the number of examples used to generate each program (median = 6 examples). The second graph shows the size of the synthesized program (median = 12 AST nodes). The third graph shows the ratio of synthesized program size to the number of examples used to generate that program (median = 1.875 seconds).

two examples distinguishing between the base case and inductive case of a data type can result in a pattern match on an appropriate value. Or, two different examples might force the synthesizer to choose an E-term involving variables and function application rather than a constant. In either case, MYTH discriminates between examples by adding complexity to the synthesized program in the form of additional language constructs.

Our hope is that our type-directed synthesis style requires a minimal amount of examples to create complex programs. To assess this, in Figure 8.3 we graph the 42 benchmark programs against the number of examples used, the size of the resulting program, and the ratio of the synthesized program to the number of examples used. These graphs better allow us to see trends in the data that we presented in table form in Figure 8.2. In particular, we see from the first graph that the average size of our synthesized programs scales with the complexity of the data types that we are operating over: bool = 6.4, nat = 10, list = 12.8, and tree = 19.75 AST nodes. However, aside from the non-recursive boolean programs whose examples completely determine their functionality (at an average of 3.6 examples per program), the second graph shows that we require roughly the same number of examples—6–7 such examples—to synthesize recursive functions over nats, lists, and trees (nat = 6.25, list = 7, and tree = 7.29).

Consequently, this means that it appears as our programs operate over more complex data types, we require comparatively fewer examples to produce more complex programs. To see this, in the third graph we compare the ratio of synthesized program size to the number of examples used to synthesize that program. While the ratios are fairly close between the bool, nat, and list programs (bool = 1.78, nat = 1.6, and tree = 1.83), the more complex tree programs benefit more with each example that we provide (tree = 2.71). The programs with the best ratios are `list_stutter` (11 AST nodes, 3 examples, 3.67 size/example ratio) and `tree_inorder` (15 AST nodes, 5 examples, 3.0 size/example ratio). In particular, the `list_stutter` we've studied throughout this work is "optimal" for list programs in the (informal) sense that we only need to present one example for each case of the list data type as well as an additional example for the Cons case to avoid synthesizing a constant. `list_stutter` also benefits greatly from I-refinement when synthesizing the $\mathsf{Cons}(x, \mathsf{Cons}(x, \blacksquare))$ portion of the Cons branch because this program fragment is entirely dictated by the two examples that are distributed to it. The program with the worst ratio is `list_drop` (13 AST nodes, 11 examples, 1.18 size/example ratio) which provides particularly tricky because we must provide several additional examples to handle the partial behavior of the function when we try to drop more elements than the length of the input list.

Synthesis Speed In addition to understanding how MYTH uses examples, we would also like to understand how quickly it can synthesize programs. Figure 8.4 shows the time taken to synthesize each program of the benchmark suite, classified

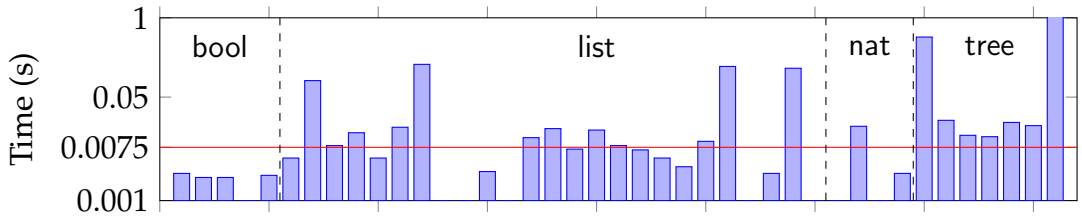


Figure 8.4: MYTH benchmark suite timing graph. Shows the execution time (averaged over five trials) to synthesize each program of the benchmark suite (median = 0.0075).

by the primary data type that the program operates over. With two exceptions, `tree_nodes_at_level` and `tree_postorder`, all the benchmark programs synthesize in sub-second time, effectively instantaneous. Looking at the average runtime of each class of programs, we see that the synthesis time scales with the complexity of the subject data type (bool = 0.00061, nat = 0.00068, list = 0.0029, and tree = 0.020 seconds). However, within each class, the run times vary wildly with a standard deviation of 0.19 seconds across all the examples. One might expect that the runtime ought to consistently scale with the size of the program, but this isn't the case because our refined synthesis procedure does not perform a breadth-first search. By effectively adopting a hybrid depth-first/breadth-first search of the refinement tree, we are able to synthesis some programs that take advantage of *I*-refinements very quickly even though they are comparatively large.

In contrast, performance stalls when we are forced to *E*-guess terms whose size crosses the exponential barrier of term generation or explore lots of nested matches. `tree_nodes_at_level` and `tree_postorder` both fall into the former category. For example, MYTH produces the following program for `tree_nodes_at_level`

```
let tree_nodes_at_level : tree -> nat -> nat =
 let rec f1 (t1:tree) : nat -> nat =
   fun (n1:nat) ->
     match t1 with
       | Leaf -> O
       | Node (t2, b1, t3) -> (match n1 with
                               | O -> S (O)
                               | S (n2) -> sum (f1 t3 n2) (f1 t2 n2))
 in
   f1
;;
```

The function application in the S branch of the inner match has size 12 which

```
type bool = True | False
type nat = O | S of nat
type list = Nil | Cons of nat * list
type cmp = LT | EQ | GT
type tree = Leaf | Node of tree * nat * tree

val andb : bool -> bool -> bool
val orb : bool -> bool -> bool
val compare : nat -> nat -> cmp
val plus : nat -> nat ->  nat
val div2 : nat -> nat
val append : list -> list -> list
```

Figure 8.5: Myth benchmarks: extended context

takes significantly longer to generate.

8.3.2 Context Size and Performance

In the previous section, we synthesized the example programs in the minimal context necessary. In practice, a synthesis tool can achieve this by allowing the user to choose which functions in scope ought to be usable during the synthesis process. However, if there are many such functions available, or the user is unsure of what functions the program might use, we will need to synthesize in a rich context. To simulate this, we also ran our benchmark suite in a context with a number of helper functions comparable to the prior work [Albarghouthi et al., 2013; Kuncak et al., 2010]. Figure 8.5 gives the signatures of the types and functions found in this context. Whenever, a benchmark program is one of the functions found in the context, we remove that function from the context so we don't end up synthesizing the trivial invocation of that top-level function.

Figure 8.6 compares the performance numbers between synthesizing the benchmark suite in a minimal context versus the extended context. It also reports the percentage difference between synthesizing in the minimal versus extended contexts. Figure 8.7 provides plots of synthesis times in the extended context along with the percentage differences in execution time between minimal and extended contexts. From the data, it is clear that the context has a very significant effect on the execution time of Myth. Overall, there is a 101% increase in performance moving from the minimal context to the extended context. Thankfully, because the synthesis times were already sub-second, the median execution time remains imperceptible at 0.0234 seconds.

However, several benchmarks experience a dramatic, noticeable change in run time. The most egregious of these examples is `list_compress` which jumps up

Test	Time (s), No Ctx	Time (s), w/ Ctx	Percent Diff
	Booleans		
bool_band	0.0028	0.0052	0.6000
bool_bor	0.0024	0.0048	0.6667
bool_impl	0.0024	0.005	0.7027
bool_neg	0.001	0.002	0.6667
bool_xor	0.0026	0.0048	0.5946
	Lists		
list_append	0.005	0.012	0.8235
list_compress	0.0932	163.4742	1.998
list_concat	0.008	0.0176	0.7500
list_drop	0.013	1.6316	1.968
list_even_parity	0.005	0.6274	1.968
list_filter	0.016	1.6792	1.962
list_fold	0.1724	0.591	1.097
list_hd	0.001	0.0234	1.836
list_inc	0.001	0.004	1.200
list_last	0.003	0.106	1.890
list_length	0.001	0.022	1.826
list_map	0.0108	0.0944	1.589
list_nth	0.0152	0.9	1.933
list_pairwise_swap	0.007	5.807	1.995
list_rev_append	0.0144	0.0314	0.7424
list_rev_fold	0.008	0.0158	0.6555
list_rev_snoc	0.0068	0.02	0.9851
list_rev_tailcall	0.005	0.0052	0.0392
list_snoc	0.0036	0.078	1.824
list_sort_sorted_insert	0.0094	0.008	-0.1609
list_sorted_insert	0.1592	27.107	1.977
list_stutter	0.001	0.0208	1.817
list_sum	0.0028	0.006	0.7273
list_take	0.1488	1.2964	1.588
list_tl	0.001	0.0216	1.823
	Natural Numbers		
nat_iseven	0.001	0.0222	1.828
nat_max	0.0166	0.0158	-0.0494
nat_pred	0.001	0.0016	0.4615
nat_add	0.0028	0.0038	0.3030
	Trees		
tree_binsert	0.4834	9.8018	1.812
tree_collect_leaves	0.0208	0.0392	0.6133
tree_count_leaves	0.0118	0.0414	1.113
tree_count_nodes	0.0112	0.035	1.030
tree_inorder	0.0192	0.0396	0.6939
tree_map	0.017	0.0362	0.7218
tree_nodes_at_level	1.1078	4.4172	1.198
tree_postorder	1.467	9.8668	1.482

Figure 8.6: MYTH benchmark suite performance results in context.

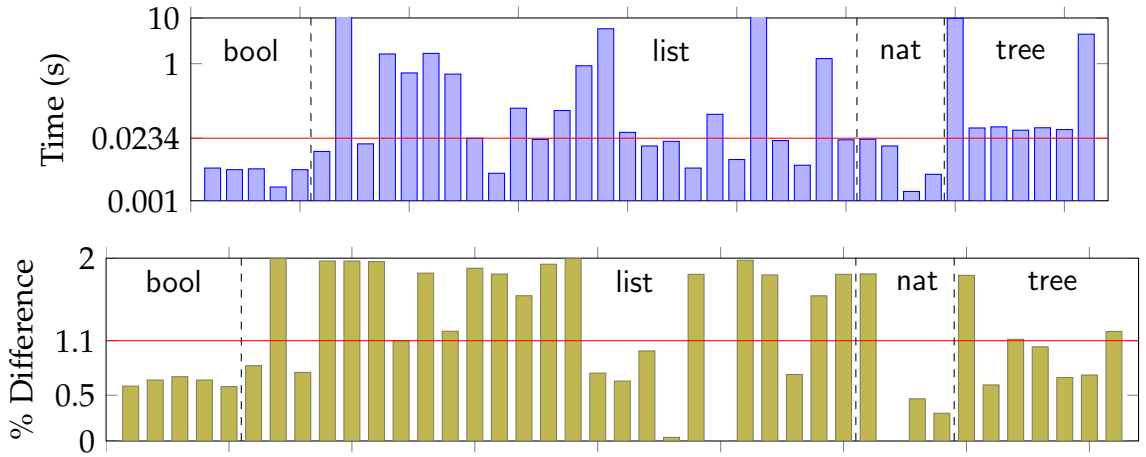


Figure 8.7: Myth benchmark suite graphs in the extended context specified by Figure 8.5 The first graph shows the execution time (in log scale) of Myth on each benchmark program in context (median = 0.0234). The second graph shows percentage difference in execution time between Myth synthesizing in a minimal context and in the extended context (median = 1.097).

to an average run time of 2.5 minutes! Examining the output for `list_compress` reveals the problem:

```
let list_compress : list -> list =
  let rec f1 (l1:list) : list =
    match l1 with
    | Nil -> Nil
    | Cons (n1, l2) ->
      (match f1 l2 with
      | Nil -> l1
      | Cons (n2, l3) ->
        (match compare n2 n1 with
        | LT -> Cons (n1, Cons (n2, l3))
        | EQ -> Cons (n1, l3)
        | GT -> Cons (n1, Cons (n2, l3))))
in
  f1
;;
```

The triply nested match is relatively quick to synthesize in the minimal context because we only have the compare function with which to make complex match scrutinees. However, with the addition of append and plus from the extended context, we now gain many more possible informative scrutinees that must be

Test	#/Examples	Prog. size	Time (s)
Arithmetic Language Interpreter			
arith	22	47	12.5408
Type Dynamic			
dyn_app_twice	6	11	0.9038
dyn_sum	25	23	24.5702
Free Variable Collector			
fvs_small	6	16	0.0372
fvs_medium	22	45	0.6782
fvs_large	31	78	3.024

Figure 8.8: MYTH extended examples results

explored in the refinement tree. The other cases where the execution time explodes, *e.g.*, `tree_binsert` and `list_sorted_insert`, have similar problem where deep matching is necessary, but the space of possible matches explodes under the extended context.

8.4 Extended Examples

In addition to the benchmark suite, we also tested MYTH on a variety of extended examples to better understand the limits of the system. Figure 8.8 gives an overview of the examples and their execution time with MYTH.

Interpreters Typed, functional programming languages are excellent for writing compilers and interpreters because abstract syntax trees line up well algebraic data types. As a result, our type-directed program synthesis style works well for synthesizing these sorts of programs. To demonstrate this, we provide an example of synthesizing an interpreter for the following calculator language:

```
type exp =
| Const of nat
| Sum of exp * exp
| Prod of exp * exp
| Pred of exp
| Max of exp * exp
```

The interpreter for this language has size 47 and requires 22 examples to synthesize in approximately 12 seconds.

```
let arith : exp -> nat =
  let rec f1 (e1:exp) : nat =
    match e1 with
      | Const (n1) -> n1
      | Sum (e2, e3) -> sum (f1 e2) (f1 e3)
      | Prod (e2, e3) -> mult (f1 e2) (f1 e3)
      | Pred (e2) -> (match f1 e2 with
                        | O -> O
                        | S (n1) -> n1)
      | Max (e2, e3) -> (match compare (f1 e2) (f1 e3) with
                           | LT -> f1 e3
                           | EQ -> f1 e3
                           | GT -> f1 e2)
  in
    f1
;;
```

It utilizes several helper functions, sum, mult, and compare, and the function itself contains multiple nested pattern matches with complex scrutinees which, from Section 8.3.1, explains why it takes significantly longer to synthesize than the benchmark suite's programs.

Type Dynamic Recall from Section 5.2.1 that part of our checks to ensure that ML_{syn} is well-founded is enforcing a positivity restriction on data types. This restrictions says that a recursive occurrence of a data type cannot appear to the left of an arrow in the type of any of its constructors. For example, consider the data type encoding type dynamic:

$$\begin{array}{l}\mathsf{type\ dyn} = \\ \quad | \ \mathsf{Error} \\ \quad | \ \mathsf{Base\ of\ nat} \\ \quad | \ \mathsf{Dyn\ of\ dyn} \to \mathsf{dyn}\end{array}$$

dyn provides dynamic typing functionality over nats (Base) and functions (Dyn). This data type breaks the positivity restriction because of the signature of Dyn, so we would not allow it in ML_{syn}.

However, as we showed in Section 5.2.1, the simplest infinite loop you can produce using this loophole requires the use of an external function (call it f : $\mathsf{dyn} \to \mathsf{dyn}$) and its subsequent double application $f\,(\mathsf{Dyn}\,f)$. Because ML_{syn} does not have let-bindings, we must synthesize a much larger program to produce this infinite loop. Furthermore, it is not clear if this is even possible to produce in ML_{syn} without the positivity restriction because we can only synthesize function applications in normal form. It is very likely that we will not encounter this

program during normal execution of Myth because if such a program with an infinite loop exists, it is likely too complicated for Myth synthesize in practice.

In light of this revelation, we left out the positivity restriction from Myth and attempted to synthesize a pair of recursive functions over dyn, `dyn_app_twice` which applies a Dyn function twice and `dyn_sum` which performs the addition operation through dyns. In either case, we produce Error if the types do not line up appropriately. In both cases we are able to synthesize the correct function, lending weight to the argument that we would not encounter infinite loops via the positivity restriction in practice. In particular, `dyn_sum` is a particular hairy function, requiring two triply nested pattern matches to correctly raise Error at all the correct points in the program.

```
let dyn_sum : dyn -> dyn -> dyn =
  fun (d1:dyn) ->
    fun (d2:dyn) ->
      match d1 with
        | Error -> Error
        | Base (n1) -> (match n1 with
                        | O -> (match d2 with
                                | Error -> Error
                                | Base (n2) -> d2
                                | Dyn (f3) -> Error)
                        | S (n2) -> (match n2 with
                                     | O -> succ d2
                                     | S (n3) -> succ (succ d2)))
        | Dyn (f3) -> Error
;;
```

Note that this pattern of programming is similar to error handling with an option/maybe type. In lieu of additional constructs to avoid excessive match expressions, we would need to use deeply nested pattern matches to crack open option/maybe values.

Free Variables Collector Myth combines depth-first and breadth-first search of the refinement tree in a particular way: first it efficiently computes the possible set of *I*-refinement all at once and then it *E*-guesses at the leaves. This makes Myth very efficient when synthesizing programs that feature shallow pattern matches over large data types. To illustrate this point, we synthesize programs that collect the free variables of a simply typed lambda calculus encoded in a locally nameless style [Aydemir et al., 2008]. The largest such program, `fvs_large`, operates over the following data type:

```
type exp =
  | Unit
  | BVar of nat
  | FVar of nat
  | Lam of nat * exp
  | App of exp * exp
  | Pair of exp * exp
  | Fst of exp
  | Snd of exp
  | Inl of exp
  | Inr of exp
  | Const of nat
  | Add of exp * exp
  | Sub of exp * exp
  | Mult of exp * exp
  | Div of exp * exp
```

Passing the necessary 31 examples to Myth produces the following function in 3.024 seconds:

```
let fvs_large : exp -> list =
  let rec f1 (e1:exp) : list =
    match e1 with
      | Unit -> []
      | BVar (n1) -> []
      | FVar (n1) -> [n1]
      | Lam (n1, e2) -> f1 e2
      | App (e2, e3) -> append (f1 e2) (f1 e3)
      | Pair (e2, e3) -> append (f1 e2) (f1 e3)
      | Fst (e2) -> f1 e2
      | Snd (e2) -> f1 e2
      | Inl (e2) -> f1 e2
      | Inr (e2) -> f1 e2
      | Const (n1) -> []
      | Add (e2, e3) -> append (f1 e2) (f1 e3)
      | Sub (e2, e3) -> append (f1 e2) (f1 e3)
      | Mult (e2, e3) -> append (f1 e2) (f1 e3)
      | Div (e2, e3) -> append (f1 e2) (f1 e3)
  in
    f1
;;
```

Note that the number of examples is necessary with our synthesis scheme because we must specify several examples per constructor to produce meaningful expressions in each branch.

While this program is larger than any that we have synthesized so far, it is not the slowest by far. This is because even though the match is tall, we discover it via I-refinement and synthesize the relatively thin branches (maximum size 9) through E-guessing very quickly. Note that we are unable to synthesize an extension of the exp data type above with a Match constructor and our naïve parameter search strategy because we would need the E-guess the function application $\mathsf{append}\ (f_1\ e_1)\ (\mathsf{append}\ (f_1\ e_2)\ (f_1\ e_3))$ of size 15, and Myth times out before it can do so. With a better tuned search strategy, synthesizing these larger branches is certainly possible for Myth

Inside-out Recursion A hallmark of a good program synthesis tool is that it can produce surprising results. To illustrate this, we close by examining a more complicated example from our benchmarks, `list_pairwise_swap`, which swaps consecutive pairs of elements in a list. When the list has odd length, we choose to return the empty list. When we provide Myth with the following set of examples,

$$\begin{aligned} &[] \Rightarrow [] \\ \mid\ &[0] \Rightarrow [] \\ \mid\ &[1] \Rightarrow [] \\ \mid\ &[1,0] \Rightarrow [0,1] \\ \mid\ &[0,1] \Rightarrow [1,0] \\ \mid\ &[0,1,0,1] \Rightarrow [1,0,1,0], \end{aligned}$$

it produces the following function:

```
let list_pairwise_swap : list -> list =
  let rec f1 (l1:list) : list =
    match l1 with
      | Nil -> []
      | Cons (n1, l2) ->
          (match l2 with
          | Nil -> []
          | Cons (n2, l3) -> Cons (n2, Cons (n1, f1 l3)))
  in
    f1
;;
```

This implementation seems to be correct as it performs the expected double pattern match, swaps the two head elements, and recursively swaps the rest. However, in the case where we call the function with the list $[1,0,1,0,1]$, it produces the list $[0,1,0,1]$, truncating the last element off the list rather than returning $[]$. To remedy this, we provide one additional example to Myth, $[1,0,1] \Rightarrow []$, and it now produces the correct result:

```
let list_pairwise_swap : list -> list =
  let rec f1 (l1:list) : list =
    match l1 with
      | Nil -> []
      | Cons (n1, l2) ->
          (match f1 l2 with
          | Nil -> (match l2 with
                    | Nil -> []
                    | Cons (n2, l3) -> Cons (n2, Cons (n1, f1 l3)))
          | Cons (n2, l3) -> [])
  in
    f1
;;
```

Rather than immediately pattern matching on the list twice, `list_pairwise_swap` instead pattern matches on *a recursive call to the function*. This seemingly odd behavior has a surprising effect: $f1\,l2$ becomes a test to see if the list has even (Nil) length or odd length (Cons)! In the case when the list has odd length, we return [], otherwise we proceed as normal.

In many situations, MYTH elects to perform this behavior, a phenomenon we call *inside-out recursion*, because it turns out to save an AST node (see `list_compress`, `list_even_parity`, and `tree_postorder`). However, it turns out this behavior is necessary to implement `list_pairwise_swap` correctly because in the absence of let-bindings or a helper function to judge whether the list has odd length, there is no other way to perform the necessary test. In this sense, even though MYTH is performing relatively naïve search, it is still able to produce results that are surprising, yet correct, at first glance!

Chapter 9

Polymorphism

One of the most glaring omissions from ML_{syn} and Myth is polymorphism. Without polymorphism, our list and tree data types from Chapter 5 must have concrete carrier types. Beyond not being able to synthesize programs that more closely resemble real-world typed, functional programs, the lack of polymorphism also impacts efficiency. To see this, consider performing raw term enumeration at the list data type (carrier type nat) with the append function of type list $\to$ list $\to$ list. Suppose that we also have variables l_1 and l_2 of type list in the context. Then we will enumerate E-terms such as:

$$\begin{gathered}\mathsf{append}\ l_1\ l_2\\ \mathsf{append}\ l_1\ (\mathsf{append}\ l_1\ l_2)\\ \mathsf{append}\ (\mathsf{append}\ l_1\ l_2)\ l_2\end{gathered}$$

As we observed in Chapter 8, functions like append pose a problem for E-term generation as they greatly increase the number of programs that we synthesize, many of which are equivalent. However, on top of these E-terms, we will also enumerate these E-terms:

$$\begin{gathered}\mathsf{append}\ l_1\ [0]\\ \mathsf{append}\ l_2\ [1]\\ \mathsf{append}\ [0]\ [0]\\ \mathsf{append}\ [1]\ [0]\\ \mathsf{append}\ [0]\ [1]\\ \mathsf{append}\ [1]\ [1].\end{gathered}$$

These E-terms are problematic because they expose the fact that the carrier type of list is concrete even though it is likely that the satisfying program does not require operating over a particular carrier type. While we are forced to explore all of these possible programs, they are all very unlikely to be satisfying the examples for this

$$
\begin{array}{lcll}
\tau & ::= & \forall\alpha.\,\tau \mid \alpha \mid \tau_1 \to \tau_2 \mid T & \text{Types} \\
e & ::= & x \mid e\,[\tau] \mid \Lambda\alpha.\,e \mid e_1\,e_2 \mid \lambda x{:}\tau.\,e \mid c & \text{Terms} \\
E & ::= & x \mid E\,[\tau] \mid E\,I & \text{Elimination Terms} \\
I & ::= & E \mid \Lambda\alpha.\,I \mid \lambda x{:}\tau.\,I \mid c & \text{Introduction Terms} \\
v & ::= & \Lambda\alpha.\,e \mid \lambda x{:}\tau.\,e \mid c & \text{Values} \\
\mathcal{E} & ::= & \Box \mid \mathcal{E}\,[\tau] \mid \mathcal{E}\,e \mid v\,\mathcal{E} & \text{Evaluation Contexts} \\
\Gamma & ::= & \cdot \mid x{:}\tau, \Gamma \mid \alpha{:}*, \Gamma & \text{Typing Contexts}
\end{array}
$$

(Extends the definition of $\lambda^{\to}_{syn}$—Figure 2.5)

$$
\boxed{e \longrightarrow e'} \qquad \textsc{eval-tapp}\ \ (\Lambda\alpha.\,e)\,[\tau] \longrightarrow [\tau/\alpha]e
$$

$$
\boxed{\begin{array}{l}\Gamma \vdash E \Rightarrow \tau \\ \Gamma \vdash I \Leftarrow \tau\end{array}}
\qquad
\textsc{t-Etapp}\ \frac{\Gamma \vdash E \Rightarrow \forall\alpha.\,\tau_1 \qquad \Gamma \vdash \tau}{\Gamma \vdash E\,[\tau] \Rightarrow [\tau/\alpha]\tau_1}
\qquad
\textsc{t-Itabs}\ \frac{\alpha{:}*, \Gamma \vdash I \Leftarrow \tau}{\Gamma \vdash \Lambda\alpha.\,I \Leftarrow \forall\alpha.\,\tau}
$$

$$
\boxed{\Gamma \vdash \tau}
\qquad
\textsc{wf-forall}\ \frac{\alpha{:}*, \Gamma \vdash \tau}{\Gamma \vdash \forall\alpha.\,\tau}
\qquad
\textsc{wf-tvar}\ \frac{\alpha{:}* \in \Gamma}{\Gamma \vdash \alpha}
\qquad
\textsc{wf-arr}\ \frac{\Gamma \vdash \tau_1 \qquad \Gamma \vdash \tau_2}{\Gamma \vdash \tau_1 \to \tau_2}
\qquad
\textsc{wf-base}\ \frac{}{\Gamma \vdash T}
$$

Figure 9.1: $\lambda^{\forall}_{syn}$ syntax and typechecking

reason.

If we had polymorphic types, then we could assign append the polymorphic type $\forall\alpha.\,\alpha\,\mathsf{list} \to \alpha\,\mathsf{list} \to \alpha\,\mathsf{list}$. With this type, we would simply be unable to synthesize E-terms like the latter set above; they would be ill-typed because the carrier type of the list is held abstract! This has the potential to be an enormous performance win as we are able to prune a large number of candidate terms from the search space when synthesizing programs over lists, trees, and other generic data types.

9.1 From System F to Polymorphic Program Synthesis

To add polymorphism to our synthesis calculi, we apply the same trick of turning the type checking rules for polymorphic values into synthesis rules. We start our exploration with System F [Girard, 1972], the simply-typed lambda calculus extended with universal quantification. Figure 9.1 gives the definition of $\lambda^{\forall}_{syn}$, our extension of System F for type-directed program synthesis. The polymorphic type $\forall\alpha.\,\tau$ possesses an introduction form, the type abstraction $\Lambda\alpha.\,e$, and an elimination form, the type application $e\,[\tau]$. For example, we would write down

the polymorphic identity function of type $\forall\alpha.\,\alpha \to \alpha$ as

$$\Lambda\alpha.\,\lambda x{:}\alpha.\,x.$$

The static and dynamic semantics of these new constructs is straightforward. Type checking a type abstraction (T-ITABS) requires that we check the abstracted type under an extended context that records the type variable α. A type application is well-typed (T-ETAPP) whenever a type abstraction of type $\forall\alpha.\,\tau_1$ is applied to a well-formed type (checked with the well-formed type judgment $\Gamma \vdash \tau$). The result of the type application is an instantiated version of the polymorphic type τ_1 where all occurrences of α have been replaced by the argument type τ. Finally, the reduction rule for type application is also simple (EVAL-TAPP) where we substitute the type argument throughout the body of the type abstraction.

9.2 Synthesizing Type Applications

With the typing rules for $\lambda^{\forall}_{syn}$ set, we can now begin the process of creating synthesis rules for polymorphic values from $\lambda^{\forall}_{syn}$'s typing judgment. Let us start with synthesizing type applications. Naïvely transforming the rule for type checking type applications, T-ETAPP, into an E-guessing rule yields:

$$\boxed{\Gamma \vdash \tau \rightsquigarrow E} \qquad \text{EGUESS-TAPP}\ \frac{\Gamma \vdash \forall\alpha.\,\tau_1 \rightsquigarrow E \qquad \Gamma \rightsquigarrow \tau}{\Gamma \vdash [\tau/\alpha]\tau_1 \rightsquigarrow E\,[\tau]}$$

A straightforward reading of this synthesis rule says that we can synthesize a type application whenever our goal type *anti-unifies* with some universal type that we can E-guess and a well-formed type that we can generate. Like the E-guessing rule for μ-types (Section 5.1), EGUESS-TAPP requires that we guess the universal type and the instantiation that will allow us to synthesize a term of the goal type through a single type application.

With recursive types, we circumvented this problem by rolling recursive types into algebraic data types and functions. And with other simple type extensions, we appealed to focusing (Section 4.1.1) to avoid guessing. However, in those cases, there were a finite number of eliminations we could perform, *i.e.*, a finite number of tuple or record components. With polymorphism, there are an infinite number of type applications we could perform because a polymorphic value can be instantiated to any type!

We can mitigate this problem by performing a breadth-first search of the polymorphic instantiations in order of increasing size of type. That is, when E-guessing, we can initially focus on the possible complete instantiations of all polymorphic variables in the context of base type (size one). For example, with

nat and bool in the context, we would explore the following instantiations of the polymorphic map function:

$$\mathsf{map}\,[\mathsf{nat}]\,[\mathsf{nat}]$$
$$\mathsf{map}\,[\mathsf{nat}]\,[\mathsf{bool}]$$
$$\mathsf{map}\,[\mathsf{bool}]\,[\mathsf{nat}]$$
$$\mathsf{map}\,[\mathsf{bool}]\,[\mathsf{bool}]$$

In successive E-guessing phases, we would expand our search to types of increasing size, *e.g.*, arrows at base types, arrows of arrows, and so forth. In the presence of many base types and polymorphic types with multiple arguments, the number of instantiations grows quickly as we must consider all the possible ways of combining these base types to fully instantiate each polymorphic type. This blows up the size of our context, but in practice, we will likely only need to explore instantiations at depth one or two (corresponding to base types and instantiated polymorphic types).

9.3 Examples for Polymorphic Values

Now, let's turn our attention towards synthesizing the polymorphic introduction form, the type abstraction. This requires that we define our example value χ for a polymorphic type. Because type abstractions are values, it is tempting to designate the type abstraction as the example value of polymorphic type, *i.e.*,

$$\chi ::= \ldots \mid \Lambda\alpha.\,\chi.$$

However, this turns out to be a bad choice of example value! To see this, imagine defining an example corresponding to the polymorphic identity function:

$$\Lambda\alpha.\,? \Rightarrow ?$$

If we were in a monomorphic setting, for example, we were defining the identity function over nats, we could write a concrete input/output pair $0 \Rightarrow 0$. But here we must fill in the question marks with values of the abstract type α, and we have no such values available!

9.3.1 Polymorphic Constants

Thus, the type abstraction value is not suitable as an example value on its own as it does not provide any values of abstract type for us to use. However, we can simply

enhance our type abstraction example value construct to provide this information!

$$\begin{array}{l} v ::= \ldots \mid q \\ \chi ::= \ldots \mid \Lambda q_1, \ldots, q_k : \alpha. \chi \mid q \end{array}$$

Rather than just introducing an abstract type α, the type abstraction example value also introduces *polymorphic constants* $q_1, \ldots, q_k$ that can be used in χ. For example, we can now specify the polymorphic identity function easily,

$$\Lambda q : \alpha. q \Rightarrow q.$$

We allow the type abstraction example value to specify multiple constants so that we can capture relationship between polymorphic values in examples. For example, suppose that we lifted our polymorphic system to ML_{syn}. Then the polymorphic stutter function might take the following examples as specification:

$$\begin{array}{l} \Lambda q_1, q_2 : \alpha. [] \Rightarrow [] \\ \qquad\qquad \mid [q_1] \Rightarrow [q_1] \\ \qquad\qquad \mid [q_2, q_1] \Rightarrow [q_2, q_2, q_1, q_1] \end{array}$$

Importantly, our polymorphic constants have decidability, identity-based equality that we employ in our compatibility check,

$$\frac{}{q \simeq q}\ \textsc{eq-pconst}$$

which allows us to distinguish between q_1 and q_2 in the above example during synthesis.

Note that our polymorphic values are not constructs that the user can synthesize.[1] You can think of them as abstract values or placeholders for polymorphic values that will be bound to variables in our example worlds' environments. For example, in the above example of the polymorphic identity function, if we synthesize the partial program:

$$\Lambda\alpha. \lambda x{:}\alpha. \blacksquare.$$

Then at the program hole, x would be bound to the polymorphic constant q in our sole example world. This means that while we can write down example values of type $\forall\alpha. \alpha$, for example, $\Lambda q : \alpha. q$, we will be unable to synthesize any terms of this type. (Which is good, because $\forall\alpha. \alpha$ is uninhabited!)

Figure 9.2 gives the enhanced syntax and synthesis rules for $\lambda^{\forall}_{syn}$. *I*-refinement of universal quantification is handled by IREFINE-FORALL where we synthesize a type abstraction. Because the examples are well-typed, they must be type abstraction example values. We assume that before synthesis (*e.g.*, during type

[1] Indeed, no rule of $\lambda^{\forall}_{syn}$ can synthesize a polymorphic constant q.

$$
\begin{array}{lcll}
v & ::= & \ldots \mid q & \text{Values} \\
\chi & ::= & \ldots \mid \Lambda q_1, \ldots, q_k : \alpha. \chi \mid q & \text{Example Values} \\
\Gamma & ::= & \cdot \mid x{:}\tau, \Gamma \mid \alpha{:}*, \Gamma \mid q{:}\alpha, \Gamma & \text{Typing Contexts}
\end{array}
$$

$\boxed{\Gamma \rightsquigarrow \tau}$

$$\textsc{wf-forall}\ \frac{\alpha{:}*, \Gamma \rightsquigarrow \tau}{\Gamma \rightsquigarrow \forall \alpha. \tau} \qquad \textsc{wf-tvar}\ \frac{\alpha{:}* \in \Gamma}{\Gamma \rightsquigarrow \alpha} \qquad \textsc{wf-arr}\ \frac{\Gamma \rightsquigarrow \tau_1 \qquad \Gamma \rightsquigarrow \tau_2}{\Gamma \rightsquigarrow \tau_1 \to \tau_2} \qquad \textsc{wf-base}\ \frac{}{\Gamma \rightsquigarrow T}$$

$\boxed{\begin{array}{c}\Gamma \vdash \tau \rightsquigarrow E \\ \Gamma \vdash \tau \rhd X \rightsquigarrow I\end{array}}$

$$\textsc{eguess-tapp}\ \frac{\Gamma \vdash \forall \alpha. \tau_1 \rightsquigarrow E \qquad \Gamma \rightsquigarrow \tau}{\Gamma \vdash [\tau/\alpha]\tau_1 \rightsquigarrow E\,[\tau]}$$

$$\textsc{irefine-forall}\ \frac{X = \overline{\Lambda q_1, \ldots, q_k{:}\alpha. \chi_i}^{i<m} \qquad X' = \overline{\chi_i}^{i<m} \qquad q_1{:}\alpha, \ldots, q_k{:}\alpha, \alpha{:}*, \Gamma \vdash \tau \rhd X' \rightsquigarrow I}{\Gamma \vdash \forall \alpha. \tau \rhd X \rightsquigarrow \Lambda \alpha. I}$$

$\boxed{v \simeq \chi}$

$$\textsc{eq-pconst}\ q \simeq q \qquad \textsc{eq-tabs}\ \frac{I \longrightarrow^* v \qquad v \simeq \chi}{\Lambda \alpha. I \simeq \Lambda q_1, \ldots, q_k{:}\alpha. \chi}$$

$\boxed{\Gamma \vdash \chi : \tau}$

$$\textsc{ex-pconst}\ \frac{q{:}\alpha \in \Gamma}{\Gamma \vdash q : \alpha} \qquad \textsc{ex-tabs}\ \frac{q_1{:}\alpha, \ldots, q_k{:}\alpha, \alpha{:}*, \Gamma \vdash \chi : \tau}{\Gamma \vdash \Lambda q_1, \ldots, q_k{:}\alpha. \chi : \forall \alpha{:}. \tau}$$

Figure 9.2: $\lambda^{\forall}_{syn}$ synthesis rules

checking), that we *normalize* the type abstraction example values so that they all declare exactly the same set of polymorphic constants. Then, example refinement simply amounts to stripping off the binders of the example values, recording them in the context, and synthesizing with the remaining nested example values. While we record the type variable associated with the polymorphic constants, this is unnecessary because we never need to refer to their types again; we merely compare the constants for equality via EX-PCONST during the synthesis process.

Finally, closing the loop on synthesis with polymorphic values, we must extend our compatibility check to type abstractions. Because we only check compatibility at well-typed program values and example values, we know that we must be comparing a type abstraction value to a type abstraction example value at universal type. (Recall that in $\lambda^{\rightarrow}_{syn}$, we separated the syntax of program values and example values, only joining them together in ML_{syn} to support partial-functions-as-program values). To do this, we simply want to check their bodies for compatibility. However, we encounter a slight complication: the body of a type abstraction program value is an arbitrary I rather than a v—in particular, the body could be an E with pending reductions. To get around this, we evaluate the body of the value type abstraction and compare the resulting value to the example value body (EQ-TABS).

As a complete example, we revisit the encoding of the boolean if function from Section 2.2.2. With polymorphic types, we can assign the boolean type and values the following lambda terms:

$$\begin{aligned}\mathsf{bool} &\stackrel{\text{def}}{=} \forall\alpha.\,\alpha \to \alpha \to \alpha\\ \mathsf{true} &\stackrel{\text{def}}{=} \Lambda\alpha.\,\lambda t{:}\alpha.\,\lambda f{:}\alpha.\,t\\ \mathsf{false} &\stackrel{\text{def}}{=} \Lambda\alpha.\,\lambda t{:}\alpha.\,\lambda f{:}\alpha.\,f\end{aligned}$$

In this setting, if has the type $\forall\alpha.\,\mathsf{bool} \to \alpha \to \alpha \to \alpha$. (Note that this type has a nested quantifier in the type synonym bool.) Our specification of the desired function is the example value:

$$\chi = \Lambda q_1, q_2{:}\alpha.\,\mathsf{true} \Rightarrow q_1 \Rightarrow q_2 \Rightarrow q_1 \;\mid\; \mathsf{false} \Rightarrow q_1 \Rightarrow q_2 \Rightarrow q_2.$$

Starting with the initial goal of

$$\blacksquare : \forall\alpha.\,\mathsf{bool} \to \alpha \to \alpha \to \alpha,$$

we apply IREFINE-FORALL to create the top-level type abstraction

$$\Lambda\alpha.\,\blacksquare : \mathsf{bool} \to \alpha \to \alpha \to \alpha,$$

where we have introduced two polymorphic constants q_1 and q_2 into the context. From here, synthesis proceeds almost identically to our $\lambda^{\rightarrow}_{syn}$ development. We

apply IREFINE-ARR multiple times to arrive a base goal type

$$\Lambda\alpha.\, \lambda b{:}\mathsf{bool}.\, \lambda t{:}\alpha.\, \lambda f{:}\alpha.\, \blacksquare : \alpha.$$

Our examples have been refined to two worlds:

$$[\mathsf{true}/b][q_1/t][q_2/f] \mapsto q_1$$
$$[\mathsf{false}/b][q_1/t][q_2/f] \mapsto q_2.$$

To guess the body of the function, we invoke IREFINE-EGUESS to guess an elimination form. We would like to synthesize the satisfying expression $b\, t\, f$ as in Chapter 2. However, because bool is now polymorphic, we must first instantiate it to the correct type, α, using EGUESS-TAPP. The complete derivation tree corresponding to this E-guessed expression is:

$$\dfrac{\dfrac{\dfrac{\dfrac{b{:}\forall\alpha.\,\alpha\to\alpha\to\alpha\in\Gamma}{\Gamma\vdash\forall\alpha.\,\alpha\to\alpha\to\alpha\rightsquigarrow b}\textsc{eguess-var} \quad \dfrac{\alpha{:}*\in\Gamma}{\Gamma\rightsquigarrow\alpha}\textsc{tguess-tvar}}{\Gamma\vdash\alpha\to\alpha\to\alpha\rightsquigarrow b\,[\alpha]}\textsc{eguess-tapp} \quad \dfrac{t{:}\alpha\in\Gamma}{\Gamma\vdash\alpha\rightsquigarrow t}\textsc{eguess-var}}{\Gamma\vdash\alpha\to\alpha\rightsquigarrow b\,[\alpha]\,t}\textsc{eguess-app} \quad \dfrac{f{:}\alpha\in\Gamma}{\Gamma\vdash\alpha\rightsquigarrow f}\textsc{eguess-var}}{\Gamma\vdash\alpha\rightsquigarrow b\,[\alpha]\,t\,f}\textsc{eguess-app}$$

9.3.2 Polymorphic Instances

The type abstraction example value that we developed in the previous section works for specifying values of polymorphic type. However, they are not the most natural way of specifying such examples. For example, rather than declaring and using abstract polymorphic constants to specify the behavior of stutter,

$$\begin{aligned}\Lambda q_1, q_2 : \alpha.\, [] &\Rightarrow [] \\ \mid [q_1] &\Rightarrow [q_1] \\ \mid [q_2, q_1] &\Rightarrow [q_2, q_2, q_1, q_1],\end{aligned}$$

it would be more convenient to specify concrete *instantiations* of stutter such as

$$\begin{aligned}[\mathsf{nat}]([] &\Rightarrow []\\ [0] &\Rightarrow [0]\\ [1,0] &\Rightarrow [1,1,0,0]),\end{aligned}$$

which is identical to the monomorphic case but an explicit type annotations for the universal type.[2]

This suggests an alternative example value for polymorphic types—the *polymorphic instance*:

$$\chi ::= \ldots \mid [\tau]\,\chi.$$

Rather than requiring the user to specify polymorphic binders, we allow them to use concrete values along with a concrete type instantiation instead. When using these concrete values, we need to remember in each example world what these type instantiations are so that we can move between type variables and concrete types as necessary. We might do this by recording *type equalities* for each example world. For example, when refining the single example world $[\mathsf{nat}]\,0 \Rightarrow 0$ for a goal type $\forall\alpha.\,\alpha \to \alpha$, we will eventually end up with the refined world

$$[\alpha\ \mathsf{nat}][0/x] \mapsto 0$$

where we remember the top-level polymorphic type variable α is instantiated to nat in this world, written $\alpha\ \mathsf{nat}$. However, the problem is that we can easily abuse this type equality to create invalid code. Say that we have this example world with the partial program skeleton $\Lambda\alpha.\lambda x{:}\alpha.\,(\blacksquare : \alpha)$. It would be unsound to use this type equality to synthesize 0 for the hole, even if this value agrees with the example, because the hole is held to an abstract type α.

Rather than recording type equalities that could apply to an entire example world, we must instead record that the example values really have polymorphic type even though they are concrete values (of a particular, known type). To do this, we introduce *boxed polymorphic expression*, written $\lceil e \rceil^{\alpha}$, that records that a particular expression ought to be treated as if had type α. These boxes behave similarly to the polymorphic constants in that they are opaque to the outside world, but allow for testable equality (as long as the carrier type of the box also has testable equality). When refining a polymorphic instance value of the form $[\tau_1]\,\chi$ at goal type $\forall\alpha.\,\tau$, we box any sub-expression of χ that would have type α when type checking χ against the polymorphic type $\forall.\,\tau$. For example, if our example value was:

$$[\mathsf{nat}]\lambda x{:}\mathsf{nat}.\,x \Rightarrow 0 \Rightarrow 1 \Rightarrow 0$$

and our goal type at this point in the synthesis process was $\forall\alpha.\,(\alpha \to \alpha) \Rightarrow \mathsf{nat} \Rightarrow \alpha \Rightarrow \alpha$. Then our boxing operation would result in the following modified example value

$$\lambda x{:}\alpha.\,\lceil x \rceil^{\alpha} \Rightarrow 0 \Rightarrow \lceil 1 \rceil^{\alpha} \Rightarrow \lceil 0 \rceil^{\alpha}.$$

Note that in addition to adding boxes in every position where we expected a value of type α according to the goal type, we also modified the lambda's type

[2] In a real implementation, we could instead infer these type annotations making the input/output examples identical to our original presentation of stutter.

annotation to reflect the fact that it is also takes an α as an argument according to the goal type. Because we are interested in determining every sub-term that has type α according to the goal type, the box transformation function **box** is the standard type checking algorithm but modified to box every such sub-term along the way.

Rather than formalize this approach completely, we observe that the instances based-approach to providing polymorphic examples is functionally equivalent to the constants-based approach we discussed in Section 9.3.1! The effect of the **box** transformation function is to take a concrete example value and abstract it according to the polymorphic goal type. And the polymorphic constants are equivalent to the boxed terms that **box** creates—they are both opaque values that we can test for equality. With this in mind, we can view polymorphic constants as the form of our example values that is simpler to reason about while polymorphic instantiations offer the user a more flexible way of specifying examples but at the cost of additional complexity.

9.4 The Metatheory of Polymorphism

We close our discussion of polymorphism by stating and proving the necessary lemmas for soundness and completeness of $\lambda^{\forall}_{syn}$.

Lemma 9.4.1 (Type Preservation of Polymorphism). *If* $\Gamma \vdash X : \forall\alpha.\,\tau$ *then* $\Gamma' \vdash X' : \tau$ *where* $\Gamma' = q_1{:}\alpha, \ldots, q_k{:}\alpha, \alpha{:}*, \Gamma$, $X = \overline{\sigma_i \mapsto \Lambda q_1, \ldots, q_k{:}\alpha.\,\chi_i}^{i<m}$ *and* $X' = \overline{\sigma_i \mapsto \chi_i}^{i<m}$.

Proof. Consider $\sigma \mapsto \Lambda q_1, \ldots, q_k{:}\alpha.\,\chi_i \in X$. Because X is well-typed, we know that σ and the type abstraction are well-typed. By inversion on EX-TABS, we know that χ is well-typed under the context $q_1{:}\alpha, \ldots, q_k{:}\alpha, \alpha{:}*, \Gamma$ which is sufficient to conclude that X' is well-typed. □

Lemma 9.4.2 (Satisfaction Soundness of Polymorphism). *If* $I \vDash X'$, *then* $\Lambda\alpha.\,I \vDash X$ *where* $X = \overline{\sigma_i \mapsto \Lambda q_1, \ldots, q_k{:}\alpha.\,\chi_i}^{i<m}$ *and* $X' = \overline{\sigma_i \mapsto \chi_i}^{i<m}$.

Proof. Consider a single $\sigma \mapsto \chi \in X'$. By the definition of satisfaction for I, $\sigma(I) \longrightarrow^* v$ and $v \simeq \chi$. Now, by unrolling the definition of satisfaction for $\Lambda\alpha.\,I$, we learn that we must show that

$$\sigma(\Lambda\alpha.\,I) = \Lambda\alpha.\,\sigma(I) \simeq \Lambda q_1, \ldots, q_k{:}\alpha.\,\chi_i.$$

However, this is immediate from EQ-TABS and the fact that $v \simeq \chi$. □

Lemma 9.4.3 (Satisfaction Preservation of Polymorphism). *If* $\Lambda\alpha.\, I \vDash X$ *then* $I \vDash X'$ *where* $X = \overline{\sigma_i \mapsto \Lambda q_1, \ldots, q_k{:}\alpha.\, \chi_i}^{i<m}$ *and* $X' = \overline{\sigma_i \mapsto \chi_i}^{i<m}$.

Proof. Consider a single $\sigma \mapsto \Lambda q_1, \ldots, q_k{:}\alpha.\, \chi_i \in X$. By the definition of satisfaction for $\Lambda\alpha.\, I$,

$$\sigma(\Lambda\alpha.\, I) = \Lambda\alpha.\, \sigma(I) \simeq \Lambda q_1, \ldots, q_k{:}\alpha.\, \chi_i.$$

Now, by unrolling the definition of satisfaction for I, we learn that we must show that $v \simeq \chi$ where $\sigma(I) \longrightarrow^* v$. However, we know this by inversion of EX-TABS on the compatibility of the type abstraction values above. □

Chapter 10

Conclusion

In this thesis, we have explored the integration of type theory into program synthesis for typed, functional programming languages. By using type theory as the basis of our synthesis techniques, we were able to:

1. Build core calculi for program synthesis rooted in the simply-typed lambda calculus and its extensions.

2. Synthesize programs that previous program synthesis systems have found difficult to address: typed, recursive functional programs over algebraic data types with higher-order functions.

3. Exploit the logical nature of types to greatly reduce the search space of possible programs, refine specification, and decompose synthesis problems into smaller, independent synthesis sub-problems.

4. Gain insight into how to synthesize programs with advanced languages features like recursion and polymorphism by inspection of the type system.

5. Leverage the power of the proof search to help optimize our search procedures in a variety of ways.

6. Reason carefully about the behavior of our core program synthesis calculi, proving soundness and completeness of the relevant synthesis procedure when possible and explaining why these properties fail when they do not hold.

In short, we have laid down a foundation for program synthesis with types and demonstrated its effectiveness. We hope that others can build upon our work to integrate types into other existing synthesis systems or begin exploring the space of program synthesis in the presence of rich types.

10.1 Future Directions

There are many areas left to explore to increase the expressiveness of the program synthesis calculi we have developed, improve the performance of MYTH, or apply these program synthesis techniques to solve problems in more targeted domains. We close by exploring these future directions in more detail.

10.1.1 Synthesis with Richer Types

The fundamental technique we introduced in this work was how to convert a standard typing judgment into a type-directed program synthesis judgment. For simple types (Chapter 4), this conversion alone was sufficient to begin synthesizing programs that use those new types. However, for more complex language features such as recursion and polymorphism, we required additional insight to integrate these features into our synthesis system.

Regardless, we have only scratched the surface of types and language features we could add to our synthesis calculi. Here are some richer types to consider as next steps and the potential challenges they present for synthesis.

Linear Types: Linear logic [Girard, 1987], linear type systems [Wadler, 1991], and other sub-structural logics [Walker, 2005] allow developers to reason about resource management policies of their programs. The heart of the linear type system is the rule for type checking (linear) variables:

$$\textsc{t-var}\ \frac{}{x{:}\tau \vdash x : \tau}$$

If we maintain the invariant that the context only contains (linear) variables interpreted as resources, this modified type checking rules states that a linear resource must be used exactly once in a program. Turning this rule into a corresponding synthesis rule is natural and straightforward. However, the difficulty lies in managing the context in the other rules to maintain our invariant. For example, when type checking pairs

$$\textsc{t-pair}\ \frac{\Gamma = \Gamma_1 \circ \Gamma_2 \qquad \Gamma_1 \vdash e_1 : \tau_1 \qquad \Gamma_2 \vdash e_2 : \tau_2}{\Gamma \vdash (e1, e2) : \tau_1 \times \tau_2}$$

we must non-deterministically partition our resources to satisfy each of the component expression of the pair. A type checking algorithm can mitigate this non-determinism by type checking e_1 first, noting what resources are used, and then type check e_2 with the remaining resources. In a synthesis system we do not know e_1 or e_2 up front, so a naíve strategy must consider all possible partitions of the contexts to generate the pair components.

Smarter search strategy and memoization techniques are required to avoid this potential combinatorial explosion of possible synthesis sub-problems.

Generalized Algebraic Data Types: Generalized Algebraic Data Types (GADTs) [Xi et al., 2003] allow for the type parameters of the return type of a constructor to vary rather than being fixed to a (polymorphic) type. The quintessential example of GADT usage is the tagless interpreter. Suppose that we have a data type corresponding to the terms of some simple language:

$$
\begin{array}{l}
\mathsf{type}\ {}'a\ \mathsf{exp} = \\
\quad \mid\ \mathsf{Const} : \mathsf{nat} \to \mathsf{nat}\ \mathsf{exp} \\
\quad \mid\ \mathsf{Pair} : {}'a\ \mathsf{exp} \to {}'b\ \mathsf{exp} \to ({}'a * {}'b)\ \mathsf{exp} \\
\ldots
\end{array}
$$

Then when we pattern match on exp values, for example, during evaluation,

$$
\begin{array}{l}
\mathsf{let\ rec\ eval}\ (e{:}{}'a\ \mathsf{exp}) : {}'a = \\
\quad \mathsf{match}\ e\ \mathsf{with} \\
\quad\quad \mid\ \mathsf{Const}\ i \to i \\
\quad\quad \ldots
\end{array}
$$

we know that the i in the Const branch of the match has type nat because our Const constructor produces a value of type ${}'a\,\mathsf{exp}$. Thus the function application $\mathsf{eval}\,(\mathsf{Const}\ 1)$ has type nat rather than some polymorphic type. This is accomplished by recording the *type equalities* that we acquire during pattern matching (${}'a \sim \mathsf{nat}$).

The primary complexity of GADTs come from the generation, propagation, and proper usage of these type equalities, in particular during type inference [Peyton Jones et al., 2006]. While a synthesis procedure does not need to directly worry about generation and propagation of these type equalities—type checking proper handles this—the synthesis procedure needs to be made aware of utilizing type equalities and anti-unification of type variables when generating terms. This is analogous to the problem of efficiently determining instantiations of polymorphic values during term generation (Section 9.2). For example, if the eval function above was in our context, our synthesis procedure needs to be able to recognize that it can produce a nat by applying eval to a Const.

Dependent Types: Finally, dependent types [Martin-Löf, 1984] allow us to express arbitrary properties of our programs that are verified during type checking. This is accomplished by allowing types to be indexed by terms of the language. For example, we might parameterize our list type to be indexed by a natural number that we intend to be the length of all lists of that type: $\mathsf{list}\,0$ is the type of the empty list, $\mathsf{list}\,3$ is the type of lists of length three, and so

forth. With this data type, we can then give our append function the richer type signature:

$$(n_1{:}\mathsf{nat}) \to (n_2{:}\mathsf{nat}) \to \mathsf{list}\, n_1 \to \mathsf{list}\, n_2 \to \mathsf{list}\, (n_1 + n_2)$$

which encodes the fact that the length of the list created by appending two lists together is the sum of their lengths.

This precision is great for program synthesis because it can dramatically reduce the search space of programs. For example, with append, there are only a few ways to produce a list of type $\mathsf{list}\, (n_1 + n_2)$ within the body of the function. However, at the same time, the search space of *types* is now much larger because it includes the space of programs!

10.1.2 Additional Specification

We have only considered input/output examples as our mode of specification (in addition to types). This is because input/output examples are amendable to refinement using the type-directed style we have developed in this thesis. However, there are many other modes of specification that we can use in tandem with examples to refine the search space of programs or reduce the number of examples we must provide to the synthesizer.

Recursive Back-patching As a concrete example, consider our approach to synthesizing recursive functions (Chapter 5). When evaluating a recursive function call, we use the input/output examples, interpreted as a partial function that specifies the behavior of the recursive function, as its value. This was because our recursive function may not be completed by synthesizing this function call, for example, if we had multiple branches of a pattern match requiring recursive function calls. But this seems wasteful; could we not use the partial function we have synthesized so far in some way?

With our normalize-and-compare strategy, we seem to be stuck without employing some kind of partial evaluation strategy. An alternative is to simply *defer* evaluation until we have a complete program to try. In the case where the candidate expression completes our recursive function, we achieve the behavior that we want. But what if we are in the troublesome situation where we have multiple branches that need to be completed, e.g.,

$$\begin{array}{l}\mathsf{match}\ x\ \mathsf{with}\\ \mid\ p_1 \to \blacksquare\\ \mid\ p_2 \to \blacksquare\\ \mid\ p_3 \to \blacksquare.\end{array}$$

Here, we must generate candidate expressions in each of the branches of the match.

Now, when we E-guess a term E in these branches, three outcomes are possible:

1. E satisfies the examples,
2. E does not satisfy the examples, or
3. E cannot be safely evaluated because it contains a recursive function call.

In the first case, we can obviously accept E as a satisfying expression. In the second case, we can safely discard E because it certainly does not contribute to an overall solution. In the third case, we need to *propagate* this expression upwards to a point where it can be evaluated safely. In the example above, this point is at the top of the match expression once we have potential candidate expressions from the other branches.

Suppose that our synthesis procedure determines that branches p_1, p_2, and p_3 have candidate expressions $\mathcal{E}_1$, $\mathcal{E}_2$, and $\mathcal{E}_3$ that could complete their respective branches. Then it is sufficient to test all combinations of potentially satisfying expressions in each branch to see if any of the combinations result in an overall satisfying program. With this approach to synthesizing recursive programs, we have to coordinate synthesis among the branches of a match rather than synthesizing them completely independently. In return, we are able to use the function currently being synthesized as part of the specification, reducing the number of examples we need to provide to the synthesizer. We call this approach to synthesizing recursive functions *recursive back-patching* because we are deferring evaluation of a recursive function until we have patched in all the points where it must make recursive calls.

First-order Constraints Another example of additional specification are first-order constraints. We have conveniently left out dealing with first-order data such as integers or strings in our type-directed approach to synthesis. This is because such data types are not inductive in nature and thus do not benefit from the proof search techniques we employ. Existing solver technology such as SMT solvers [Barrett et al., 2008] deal with constraints well; can we integrate both forms of specification into our synthesis algorithm?

Consider synthesizing a pattern match,

$$
\begin{array}{l}
\mathsf{let}\ n : \mathsf{int} = \blacksquare\ \mathsf{in} \\
\quad \mathsf{match}\ x\ \mathsf{with} \\
\quad \mid\ p_1 \rightarrow \blacksquare \\
\quad \mid\ p_2 \rightarrow \blacksquare \\
\quad \mid\ p_3 \rightarrow \blacksquare,
\end{array}
$$

whose branches will perform operations over the integer n that we have yet to synthesize. Each branch will generate constraints on n, *e.g.*, inequalities such as $n > 3$, that will constrain what n can be. We cannot solve those constraints

locally in the branches, we must *propagate* those constraints back to the definition of n. At this point, we can discharge those constraints to a solver to generate an appropriate n. In the case where no such n can be generated from the constraints, we know that our choices of sub-expressions in the branches are bad, and we must revise them somehow, for example by using counter-examples to help refine the program [Solar-Lezama, 2008].

Push-down versus Bubble-up Evidence Recursive back-patching and first-order constraints seem like unrelated modes of specification. However, from our discussion above, we can tell that they are actually related! They are both forms of what we call "bubble-up" evidence where we generate information at the leaves of a synthesis derivation—potential programs or constraints—and must propagate that information upwards in the derivation to a point where we can utilize that information correctly—evaluating a complete program or invoking a solver. In contrast, the example refinement mechanisms we have developed in this thesis work in the opposite direction: they decompose examples and push information downward into the leaves of the synthesis derivation. Thus, characterizing specification or evidence by their flow in the synthesis procedure, push-down or bubble-up, gives us important insight into how we might integrate both forms of specification into a robust synthesis system.

10.1.3 Enumeration Modulo Equivalences

From Chapter 8, we see that the largest bottleneck in a practical synthesis system based on our type-and-example driven style is E-guessing. Back in Chapter 2, we demonstrated that the search space of programs grows exponentially with the size of the programs under consideration (Figure 2.2). Virtually all of the optimizations we have considered were designed to keep us from falling off this "exponential cliff" where it becomes infeasible to enumerate E-terms. We can consider additional ways of refining examples to push the cliff further away, *e.g.*, asserting particular example-refining axioms about known functions [Feser et al., 2015], but we still have the fundamental problem of generating terms more efficiently.

One large area of optimization we did not consider in this work is enumerating programs modulo equivalence classes. We already do this to some degree, for example, syntactically restricting synthesized programs to be in normal form, immediately ruling out all non-normal programs in the process. However, there are many other ways to derive equivalence classes such as employing congruence closure [Nelson and Oppen, 1980], recording evaluation results [Albarghouthi et al., 2013], or user-defined equivalences [Feser et al., 2015]. The latter form of equivalences classes are important to consider because user-defined types and operations carry their own equational reasoning principles. For example, addition-by-zero of natural numbers or appending empty lists are both equivalences that

depend on the data types (nat and list) and the behavior of the operations (plus and append) involved. We might even want to derive these equations automatically based on observation during the synthesis process, specialized to the examples that we are synthesizing over.

Nevertheless, there are many opportunities for employing reasoning about equivalence to cut down on the search space of programs, and it is very likely that no single technique will prove to be a silver bullet to this problem. For synthesis, we must consider an appropriate combination of techniques that allow us to scale up our technology to larger and more complex programs.

10.1.4 Applications of Program Synthesis with Types

With the foundations of type-directed program synthesis set, a final question worth asking is "What is the practical use of this technology?" Certainly, we will likely never reach the point where we can synthesize all of our programs from input/output examples or other forms of specification. So what applications of type-directed program synthesis are more realistic to aim for?

Programming Assistance Tools Rather than trying to synthesize a program from scratch, can we involve the user in meaningful ways to guide the synthesis process or provide alternative forms of synthesis output such as visualization or code skeletons (such as those provided by the Leon synthesis tool for Scala programs [Kneuss et al., 2013]) to help the developer? Our refinement tree data structure offers promising insight into this area. Recall that the refinement tree represents all of the decisions about the design of a program that we can derive from the examples given by the user. By prioritizing particular branches of the refinement tree using heuristics, we can present the code skeletons to the user that are most likely to lead to a satisfying program. For example, consider the arithmetic language interpreter that we presented in Section 8.4. Suppose that rather than trying to generate a completed program, we instead presented the code skeleton:

```
let rec arith (e:exp) : nat =
  match e with
  | Const (n1) -> n1
  | Sum (e2, e3) -> ??
  | Prod (e2, e3) -> ??
  | Pred (e2) -> (match f1 e2 with
                  | O -> O
                  | S (n1) -> ??)
  | Max (e2, e3) -> (match compare (f1 e2) (f1 e3) with
                     | LT -> ??
                     | EQ -> ??
                     | GT -> ??)
```

where the user was tasked to fill in the *E*-guessed holes themselves armed with the knowledge of the input/output examples they need to satisfy at those positions. The code skeleton is relatively quick to complete because it is dictated by *I*-refinements, but still has the potential to provide a lot of value to the programmer as the problem of designing the complex interpreter has been distilled into filling in a number of smaller holes.

From Enforcement to Synthesis Finally, let us come full circle on the ideas that we developed at the beginning of this work. We observed that the primary role of rich type systems was to enforce increasingly complex properties of programs, for example concurrency protocols [Mazurak and Zdancewic, 2010], information flow properties [Jia et al., 2008], differential privacy [Gaboardi et al., 2013], and other domain-specific concerns [Hudak, 1998], among others. However, the same mechanisms that enforce properties of programs also allow us to constrain the space of possible programs, so we were able to appropriate type systems towards the goal of synthesizing programs. A natural end-goal of this work, therefore, is to turn type systems designed to enforce particular properties of programs into program synthesizers that synthesize programs who possess these properties automatically by construction!

Currently, the types that we are able to handle efficiently in our program synthesis tools are too weak to handle these more advanced properties. But by designing synthesis strategies around richer classes of types, we can piggy back on top of the large body of existing work on type systems to synthesize more relevant and interesting programs. This final goal is perhaps the most ambitious, but also the most rewarding if we can achieve it: for every interesting, specialized type system that we develop, we can build a related synthesizer that builds programs for that type system automatically. In this ideal world, program synthesis and types enjoy a close, synergistic relationship where types help build programs and synthesizers help make types even more relevant and useful!

Appendix A

The Implementation of Myth

In Chapter 7 we evolved the synthesis judgment of ML_{syn} into an efficient synthesis procedure. This procedure went through several iterations in the chapter, so we summarize the final product here.

The synthesis procedure that we implement directly in our prototype program synthesis tool, Myth (Chapter 8), consists of:

- A *refinement tree generation algorithm* (Figure A.1),
- A *relevant term generation algorithm* (Figure A.2 and Figure A.3), and
- An overall synthesis procedure that, given a synthesis specification, synthesizes the first program it finds that satisfies that specification.

Given a signature Σ, context Γ, goal type τ, and examples X, our synthesis procedure operates as follows:

1. Let our search parameter initial values be $s = 1$ (the maximum match scrutinee size), $m = 0$ (the maximum match depth), and $k = 13$ (the maximum E-term size).
2. Starting with the initial synthesis I-refinement phase of the synthesis plan described below:
   (a) Execute the next phase of I-refinement.
   (b) Perform E-guessing for 0.25 seconds (using **guess** defined in Figure A.1) up to size k at all elligible nodes in the current refinement tree. A refinement tree node is elligible for E-guessing if we have not yet found a satisfying expression at that node and its corresponding goal type is a base type T.
   (c) Propogate newly found satisfying programs upwards in the refinement tree. If we construct an overall, satisfying program to the original synthesis problem then we terminate and return that program. Otherwise, we move onto the next phase of I-refinement and repeat.

The phases of I-refinement in our synthesis plan are:

1. Create an initial refinement tree with the search parameters s and m.
2. Increase m by one ($m = 1$) and extend the refinement tree accordingly.
3. Increase m by one ($m = 2$) and extend the refinement tree accordingly.
4. Increase s by five ($s = 6$) and extend the refinement tree accordingly.
5. Increase m by one ($m = 3$).

If we are unable to find a satisfying program and run out of synthesis steps, we return an error stating that the procedure could not find a satisfying program.

$$\mathbf{types}(\Sigma) = \{T \mid C : \tau_1 * \ldots * \tau_m \to T \in \Sigma\}$$
$$\mathbf{rmatch}^s(\Sigma;\Gamma;\tau;X;0) = \{\}$$
$$\mathbf{rmatch}^s(\Sigma;\Gamma;\tau;X;k) = \bigcup_{T \in \mathbf{types}(\Sigma)} \{\underline{\mathsf{match}\ E\ \mathsf{with}\ \overline{p_i \to \blacksquare_i}^{i<m}} \mid$$
$$E \in \mathbf{gen}_E(\Sigma;\Gamma;T;s),$$
$$\overline{(p_i, X_i)}^{i<m} = \mathbf{distribute}(\Sigma, T, X, E),$$
$$\overline{\Gamma_i}^{i<m} = \mathbf{binders}(\Gamma, E, p_i),$$
$$\overline{\blacksquare_i = \mathbf{rtree}^s(\Sigma;\Gamma_i,\Gamma;\tau;X_i;k-1)}^{i<m}\}$$

$$\mathbf{rtree}^s(\Sigma;\Gamma;\tau_1 \to \tau_2;X;k) = \{\underline{\mathsf{fix}\ f\ (x{:}\tau_1) : \tau_2 = \blacksquare} \mid$$
$$X' = \mathbf{apply}(f, x, \sigma_1 \mapsto \rho_1) +\!\!+ \ldots +\!\!+ \mathbf{apply}(f, x, \sigma_n \mapsto \rho_n),$$
$$\blacksquare = \mathbf{rtree}^s(\Sigma; f{:}\tau_1 \to \tau_2\{\mathsf{rec}\}, x{:}\tau_1\{\mathsf{arg}\ f\};\Gamma;X';k)$$
$$\}$$
where
$$X = \sigma_1 \mapsto \rho_1, \ldots, \sigma_n \mapsto \rho_n$$

$$\mathbf{rtree}^s(\Sigma;\Gamma;T;X;k) = \{\underline{C(\blacksquare_1, \ldots, \blacksquare_m)} \mid$$
$$X_1, \ldots, X_m = \mathbf{proj}(X),$$
$$\overline{\blacksquare_i \in \mathbf{rtree}^s(\Sigma;\Gamma;\tau_i;X_i;k)}^{i<m}$$
$$\} \cup \mathbf{rmatch}^s(\Sigma;\Gamma;\tau;X;k)$$
where
$$X = \overline{\sigma_i \mapsto C(I_{i1}, \ldots, C_{im})}^{i<n}$$
$$C : \tau_1 * \ldots * \tau_m \in \Sigma$$

$$\mathbf{guess}(\Sigma;\Gamma;\tau;X;0) = \{\}$$
$$\mathbf{guess}(\Sigma;\Gamma;\tau;X;k) = \{E \mid E \in \mathbf{gen}_E(\Sigma;\Gamma;\tau;k), E \vDash X\}$$

Figure A.1: Refinement tree creation and E-guessing

$$\boxed{\mathbf{gen}_E(\Sigma;\Gamma;\tau;n)}$$

$$\begin{aligned}
\mathbf{gen}_E(\Sigma;\cdot;\tau;n) &= \{\} \\
\mathbf{gen}_E(\Sigma;\cdot;\tau;0) &= \{\} \\
\mathbf{gen}_E(\Sigma;x{:}\tau_1,\Gamma;\tau;n) &= \mathbf{gen}_E^{x:\tau_1}(\Sigma;\Gamma;\tau;n) \cup \mathbf{gen}_E(\Sigma;\Gamma;\tau;n)
\end{aligned}$$

$$\boxed{\mathbf{gen}_E^{x:\tau_1}(\Sigma;\Gamma;\tau;n)}$$

$$\begin{aligned}
\mathbf{gen}_E^{x:\tau_1}(\Sigma;\Gamma;\tau;0) &= \{\} \\
\mathbf{gen}_E^{x:\tau}(\Sigma;\Gamma;\tau;1) &= \{x\} \\
\mathbf{gen}_E^{x:\tau_1}(\Sigma;\Gamma;\tau;1) &= \{\} \qquad (\tau \neq \tau_1) \\
\mathbf{gen}_E^{x:\tau_1}(\Sigma;\Gamma;\tau;n) &= \bigcup_{\tau_2\to\tau\in\Gamma} \bigcup_{k=1}^{n-1} \\
&\quad (\mathbf{gen}_E^{x:\tau_1}(\Sigma;\Gamma;\tau_2\to\tau;k) \underset{app}{\otimes} \mathbf{gen}_I(\Sigma;\Gamma;\tau_2;n-k)) \\
&\cup\ (\mathbf{gen}_E(\Sigma;\Gamma;\tau_2\to\tau;k) \underset{app}{\otimes} \mathbf{gen}_I^{x:\tau_1}(\Sigma;\Gamma;\tau_2;n-k)) \\
&\cup\ (\mathbf{gen}_E^{x:\tau_1}(\Sigma;\Gamma;\tau_2\to\tau;k) \underset{app}{\otimes} \mathbf{gen}_I^{x:\tau_1}(\Sigma;\Gamma;\tau_2;n-k))
\end{aligned}$$

Figure A.2: Relevant E-term generation

$$\boxed{\mathbf{gen}_I(\Sigma, \Gamma, \tau, n)}$$

$$\begin{aligned}
\mathbf{gen}_I(\Sigma; \cdot; \tau; n) &= \{\} \\
\mathbf{gen}_I(\Sigma; \Gamma; \tau; 0) &= \{\} \\
\mathbf{gen}_I(\Sigma; x : \tau_1, \Gamma; \tau; n) &= \mathbf{gen}_I^{x:\tau_1}(\Sigma; \Gamma; \tau; n) \cup \mathbf{gen}_I(\Sigma; \Gamma; \tau; n) \\
\mathbf{gen}_I(\Sigma; \cdot; \tau_1 \to \tau_2; n) &= \\
&\{\mathsf{fix}\ f\ (x{:}\tau_1) : \tau_2 = I \mid I \in \mathbf{gen}_I(\Sigma; f : \tau_1 \to \tau_2, x : \tau_1; \tau_2; n-1)\}
\end{aligned}$$

$$\mathbf{gen}_I^{x:\tau}(\Sigma; \Gamma; T; n) = \mathbf{gen}_E^{x:\tau}(\Sigma; \Gamma; T; n)$$
$$\bigcup_{C:\tau_1 * \ldots * \tau_k \to T \in \Sigma} \quad \bigcup_{\substack{n_1,\ldots,n_k \text{ for} \\ n_1+\ldots+n_k=n}}$$
$$\{C(I_1, \ldots, I_k) \mid I_j \in \mathbf{gen}_I(\Sigma; \Gamma; \tau_j; n_j)\}$$

$$\boxed{\mathbf{gen}_I^{x:\tau_1}(\Sigma; \Gamma; \tau; n)}$$

$$\begin{aligned}
\mathbf{gen}_I^{x:\tau}(\Sigma; \Gamma; \tau_1 \to \tau_2; n) &= \mathbf{gen}_E^{x:\tau}(\Sigma; \Gamma; \tau_1 \to \tau_2; n) \cup \\
&\{\mathsf{fix}\ f\ (y{:}\tau_1) : \tau_2 = I \mid I \in \mathbf{gen}_I^{x:\tau}(\Sigma; f : \tau_1 \to \tau_2, y : \tau_1, \Gamma; \tau_2; n-1)\}
\end{aligned}$$

$$\mathbf{gen}_I^{x:\tau}(\Sigma; \Gamma; T; n) = \mathbf{gen}_E^{x:\tau}(\Sigma; \Gamma; T; n)$$
$$\bigcup_{C:\tau_1 * \ldots * \tau_k \to T \in \Sigma} \quad \bigcup_{\substack{n_1,\ldots,n_k \text{ for} \\ n_1+\ldots+n_k=n}} \quad \bigcup_{\substack{r_1,\ldots,r_k \in \\ \mathbf{parts}(k)}}$$
$$\{C(I_1, \ldots, I_k) \mid I_j \in \mathbf{genp}_I^{m_j;x:\tau}(\Sigma; \Gamma; \tau_j; n_j)\}$$

$$\mathbf{parts}(k) = \{\underbrace{\mathsf{Not}, \ldots, \mathsf{Not}}_{i-1}, \mathsf{Must}, \underbrace{\mathsf{May}, \ldots, \mathsf{May}}_{k-i} \mid i \in 1, \ldots, k\}$$

$$\mathbf{genp}_I^{r;x:\tau_1}(\Sigma; \Gamma; \tau; n) = \begin{cases} \mathbf{gen}_I^{x:\tau_1}(\Sigma; \Gamma; \tau; n) & r = \mathsf{Must} \\ \mathbf{gen}_I(\Sigma; x : \tau_1, \Gamma; \tau; n) & r = \mathsf{May} \\ \mathbf{gen}_I(\Sigma; \Gamma; \tau; n) & r = \mathsf{Not} \end{cases}$$

Figure A.3: Relevant I-term generation

Appendix B

The Myth Test Suite

In Chapter 8, we analyzed Myth over a variety of benchmark programs. In this chapter, we present the full source code these benchmarks as well as the output of Myth on these benchmarks. The current code for Myth is freely available from the author's website.

B.1 Contexts

`tests/pldi-2015-benchmarks/bool.decls`

```
type bool =
  | True
  | False
```

`tests/pldi-2015-benchmarks/nat.decls`

```
type nat =
  | O
  | S of nat

type bool =
  | True
  | False
```

`tests/pldi-2015-benchmarks/compare.decls`

```
type cmp =
  | LT
  | EQ
  | GT
```

```
let rec compare (n1:nat) (n2:nat) : cmp =
  match n1 with
  | O ->
    (match n2 with
    | O -> EQ
    | S (m) -> LT)
  | S (m1) ->
    (match n2 with
    | O -> GT
    | S (m2) -> (compare m1 m2))
;;
```

---

tests/pldi-2015-contexts/context.decls

```
type bool =
  | True
  | False

let rec andb (n1:bool) (n2:bool) : bool =
  match n1 with
  | True -> n2
  | False -> False
;;

let rec orb (n1:bool) (n2:bool) : bool =
  match n1 with
  | True -> True
  | False -> n2
;;

type nat =
  | O
  | S of nat

let rec plus (n1:nat) (n2:nat) : nat =
  match n1 with
  | O -> n2
  | S (n1) -> S (plus n1 n2)
;;

let rec div2 (n:nat) : nat =
  match n with
  | O -> O
  | S (n1) -> match n1 with
    | O -> O
    | S (n2) -> S (div2 n2)
;;

type list =
  | Nil
```

```
  | Cons of nat * list

let rec append (l1:list) (l2:list) : list =
  match l1 with
  | Nil -> l2
  | Cons (x, l1) -> Cons (x, append l1 l2)
;;

type cmp =
  | LT
  | EQ
  | GT

let rec compare (n1 : nat) (n2 :nat) : cmp =
  match n1 with
  | O -> (match n2 with
            | O -> EQ
            | S (m) -> LT
          )
  | S (m1) ->
      ( match n2 with
      | O -> GT
      | S (m2) -> (compare m1 m2) )
;;

type tree =
  | Leaf
  | Node of tree * nat * tree
```

B.2 Benchmark Suite

tests/pldi-2015-benchmarks/bool_band.ml

```
#use "bool.decls"

let bool_band : bool -> bool -> bool |>
  { True => True => True
  ; True => False => False
  ; False => True => False
  ; False => False => False } = ?
```

```
let bool_band : bool -> bool -> bool =
  fun (b1:bool) -> fun (b2:bool) -> match b1 with
                                      | True -> b2
                                      | False -> False
;;
```

tests/pldi-2015-benchmarks/bool_bor.ml

```
#use "bool.decls"

let bool_bor : bool -> bool -> bool |>
  { True => True => True
  ; True => False => True
  ; False => True => True
  ; False => False => False } = ?
```

```
let bool_bor : bool -> bool -> bool =
  fun (b1:bool) -> fun (b2:bool) -> match b1 with
                                      | True -> True
                                      | False -> b2
;;
```

tests/pldi-2015-benchmarks/bool_impl.ml

```
#use "bool.decls"

let bool_impl : bool -> bool -> bool |>
  { True => True => True
  ; True => False => False
  ; False => True => True
  ; False => False => True } = ?
```

```
let bool_impl : bool -> bool -> bool =
  fun (b1:bool) -> fun (b2:bool) -> match b1 with
                                      | True -> b2
                                      | False -> True
;;
```

tests/pldi-2015-benchmarks/bool_neg.ml

```
#use "bool.decls"

let bool_neg : bool -> bool |>
  { True => False
  ; False => True } = ?
```

```
let bool_neg : bool -> bool =
  fun (b1:bool) -> match b1 with
                     | True -> False
                     | False -> True
;;
```

tests/pldi-2015-benchmarks/bool_xor.ml

```
#use "bool.decls"

let bool_xor : bool -> bool -> bool |>
  { True => True => False
  ; True => False => True
  ; False => True => True
  ; False => False => False } = ?
```

```
let bool_xor : bool -> bool -> bool =
  fun (b1:bool) ->
    fun (b2:bool) ->
      match b1 with
        | True -> (match b2 with
                     | True -> False
                     | False -> True)
        | False -> b2
;;
```

tests/pldi-2015-benchmarks/list_append.ml

```
type nat =
  | O
  | S of nat

type list =
  | Nil
  | Cons of nat * list

let list_append : list -> list -> list |>
  { [] => ( [] => []
          | [0] => [0])
  | [0] => ( [] => [0]
           | [0] => [0; 0])
  | [1;0] => ( [] => [1; 0]
             | [0] => [1; 0; 0])
  } = ?
```

```
let list_append : list -> list -> list =
  let rec f1 (l1:list) : list -> list =
    fun (l2:list) ->
      match l1 with
        | Nil -> l2
        | Cons (n1, l3) -> Cons (n1, f1 l3 l2)
  in
    f1
;;
```

tests/pldi-2015-benchmarks/list_compress.ml

```
type nat =
  | O
  | S of nat

type list =
  | Nil
  | Cons of nat * list

type cmp =
  | LT
  | EQ
  | GT

let rec compare (n1 : nat) (n2 :nat) : cmp =
  match n1 with
  | O -> (match n2 with
           | O -> EQ
           | S (m) -> LT
         )
  | S (m1) ->
      ( match n2 with
      | O -> GT
      | S (m2) -> (compare m1 m2) )
;;

let list_compress : list -> list |>
{   [] => []
  | [0] => [0]
  | [1] => [1]
  | [0;0] => [0]
  | [1;1] => [1]
  | [2;0] => [2;0]
  | [1;0;0] => [1;0]
  | [0;1;1] => [0;1]
  | [2;1;0;0] => [2;1;0]
  | [2;2;1;0;0] => [2;1;0]
  | [2;2;0] => [2;0]
  | [2;2;2;0] => [2;0]
  | [1;2;2;2;0] => [1;2;0]
} = ?
```

```
let list_compress : list -> list =
  let rec f1 (l1:list) : list =
    match l1 with
      | Nil -> Nil
      | Cons (n1, l2) -> (match f1 l2 with
                          | Nil -> l1
                          | Cons (n2, l3) -> (match compare n2 n1 with
```

```
                                                  | LT -> Cons (n1,
                                                            Cons (n2, l3))
                                                  | EQ -> Cons (n1, l3)
                                                  | GT -> Cons (n1,
                                                            Cons (n2, l3))))
  in
    f1
;;
```

tests/pldi-2015-benchmarks/list_concat.ml

```
type nat =
  | O
  | S of nat

type list =
  | Nil
  | Cons of nat * list

type llist =
  | LNil
  | LCons of list * llist

let rec append (l1:list) (l2:list) : list =
  match l1 with
  | Nil -> l2
  | Cons (x, l1p) -> Cons (x, append l1p l2)
;;

let list_concat : llist -> list |>
  { LNil => []
  | LCons ([], LNil) => []
  | LCons ([0], LNil) => [0]
  | LCons ([0], LCons([0], LNil)) => [0;0]
  | LCons ([1], LNil) => [1]
  | LCons ([1], LCons([1], LNil)) => [1;1]
  } = ?
```

```
let list_concat : llist -> list =
  let rec f1 (l1:llist) : list =
    match l1 with
      | LNil -> Nil
      | LCons (l2, l3) -> append l2 (f1 l3)
  in
    f1
;;
```

tests/pldi-2015-benchmarks/list_drop.ml

```
type nat =
  | O
  | S of nat

type list =
  | Nil
  | Cons of nat * list

let list_drop : list -> nat -> list |>
  { [] => ( 0 => []
          | 1 => [] )
  | [0] => ( 0 => [0]
           | 1 => [] )
  | [1] => ( 0 => [1]
           | 1 => [] )
  | [1; 0] => ( 0 => [1; 0]
              | 1 => [0] )
  | [0; 1] => ( 0 => [0; 1]
              | 1 => [1]
              | 2 => [] )
  } = ?
```

```
let list_drop : list -> nat -> list =
  let rec f1 (l1:list) : nat -> list =
    fun (n1:nat) ->
      match n1 with
        | O -> l1
        | S (n2) -> (match l1 with
                       | Nil -> Nil
                       | Cons (n3, l2) -> f1 l2 n2)
  in
    f1
;;
```

tests/pldi-2015-benchmarks/list_even_parity.ml

```
type nat =
  | O
  | S of nat

type bool =
  | True
  | False

type list =
  | Nil
  | Cons of bool * list
```

```
let list_even_parity : list -> bool |>
    { [] => True
    | [ False ] => True
    | [ True  ] => False
    | [ False ; False ] => True
    | [ False ; True ] => False
    | [ True  ; False ] => False
    | [ True  ; True ] => True
    } = ?
```

```
let list_even_parity : list -> bool =
  let rec f1 (l1:list) : bool =
    match l1 with
      | Nil -> True
      | Cons (b1, l2) -> (match f1 l2 with
                          | True -> (match b1 with
                                     | True -> False
                                     | False -> True)
                          | False -> b1)
  in
    f1
;;
```

---

tests/pldi-2015-benchmarks/list_filter.ml

```
type nat =
  | O
  | S of nat

type list =
  | Nil
  | Cons of nat * list

type bool =
  | True
  | False

let rec is_even (n:nat) : bool =
  match n with
  | O -> True
  | S (n1) ->
    match n1 with
    | O -> False
    | S (n2) -> is_even n2
;;

let rec is_nonzero (n:nat) : bool =
  match n with
  | O -> False
```

```
  | S (n1) -> True
;;

let list_filter : (nat -> bool) -> list -> list |>
{
  is_even => ( [] => []
             | [0] => [0]
             | [1] => []
             | [2] => [2]
             | [0;0] => [0;0]
             | [0;1] => [0] )
| is_nonzero => ( [] => []
                | [0] => [] )
} = ?
```

```
let list_filter : (nat -> bool) -> list -> list =
  fun (f2:nat -> bool) ->
    let rec f3 (l1:list) : list =
      match l1 with
        | Nil -> Nil
        | Cons (n1, l2) -> (match f2 n1 with
                            | True -> Cons (n1, f3 l2)
                            | False -> Nil)
    in
      f3
;;
```

---

tests/pldi-2015-benchmarks/list_fold.ml

```
type nat =
  | O
  | S of nat

type bool =
  | True
  | False

type list =
  | Nil
  | Cons of nat * list

let rec sum (n1:nat) (n2:nat) : nat =
  match n1 with
  | O -> n2
  | S (n1) -> S (sum n1 n2)
;;

let rec is_odd (n:nat) : bool =
  match n with
```

```
  | O -> False
  | S (n) ->
    (match n with
    | O -> True
    | S (n1) -> is_odd n1)
;;

let count_odd : nat -> nat -> nat =
  fun (n1:nat) -> fun (n2:nat) ->
    match is_odd n2 with
    | True -> S (n1)
    | False -> n1
;;

let list_fold : (nat -> nat -> nat) -> nat -> list -> nat |>
    { sum => ( 0 => ( [] => 0
                    | [1] => 1
                    | [2; 1] => 3
                    | [3; 2; 1] => 6 )
             | 1 => [] => 1 )
    | count_odd => ( 0 => ( [] => 0
                          | [1] => 1
                          | [2; 1] => 1
                          | [3; 2; 1] => 2 ) )
    } = ?
```

```
let list_fold : (nat -> nat -> nat) -> nat -> list -> nat =
  fun (f2:nat -> nat -> nat) ->
    fun (n1:nat) ->
      let rec f4 (l1:list) : nat =
        match l1 with
          | Nil -> n1
          | Cons (n2, l2) -> f2 (f4 l2) n2
      in
        f4
;;
```

---

tests/pldi-2015-benchmarks/list_hd.ml

```
type nat =
  | O
  | S of nat

type list =
  | Nil
  | Cons of nat * list

let list_hd : list -> nat |>
  { [] => 0
```

```
  | [0] => 0
  | [1] => 1
  } = ?
```

```
let list_hd : list -> nat =
  fun (l1:list) -> match l1 with
                   | Nil -> O
                   | Cons (n1, l2) -> n1
;;
```

tests/pldi-2015-benchmarks/list_inc.ml

```
type nat =
  | O
  | S of nat

type list =
  | Nil
  | Cons of nat * list

let rec map (l:list) (f : nat -> nat) : list =
     match l with
       | Nil -> Nil
       | Cons (n, ls) -> Cons (f n, map ls f)
;;

let list_inc : list -> list |>
  { [] => []
  | [1;2] => [2;3]
  | [0;0] => [1;1]
  | [3;4;5] => [4;5;6]
  } = ?
```

```
let list_inc : list -> list =
  fun (l1:list) -> map l1 (fun (n1:nat) -> S (n1))
;;
```

tests/pldi-2015-benchmarks/list_last.ml

```
type nat =
  | O
  | S of nat

type list =
  | Nil
  | Cons of nat * list
```

```
type natopt =
  | None
  | Some of nat

let list_last : list -> natopt |>
  { [] => None
  | [1] => Some (1)
  | [2] => Some (2)
  | [2; 1] => Some (1)
  | [1; 2] => Some (2)
  | [3; 2; 1] => Some (1)
  } = ?
```

```
let list_last : list -> natopt =
  let rec f1 (l1:list) : natopt =
    match l1 with
      | Nil -> None
      | Cons (n1, l2) -> (match l2 with
                          | Nil -> Some (n1)
                          | Cons (n2, l3) -> f1 l2)
  in
    f1
;;
```

tests/pldi-2015-benchmarks/list_length.ml

```
type nat =
  | O
  | S of nat

type list =
  | Nil
  | Cons of nat * list

let list_length : list -> nat |>
  { [] => O
  | [O] => S O
  | [O;O] => S (S O) } = ?
```

```
let list_length : list -> nat =
  let rec f1 (l1:list) : nat =
    match l1 with
      | Nil -> O
      | Cons (n1, l2) -> S (f1 l2)
  in
    f1
;;
```

tests/pldi-2015-benchmarks/list_map.ml

```
type nat =
  | O
  | S of nat

type list =
  | Nil
  | Cons of nat * list

let zero (n:nat) : nat = O ;;
let inc (n:nat): nat = S (n) ;;

let list_map : (nat -> nat) -> list -> list |>
  { inc => ( [] => []
           | [0] => [1]
           | [0; 0] => [1; 1]
           | [1] => [2]
           | [1; 1] => [2; 2] )
  | zero => ( [] => []
            | [0] => [0]
            | [0; 0] => [0; 0] )
  } = ?
```

```
let list_map : (nat -> nat) -> list -> list =
  fun (f2:nat -> nat) ->
    let rec f3 (l1:list) : list =
      match l1 with
        | Nil -> Nil
        | Cons (n1, l2) -> Cons (f2 n1, f3 l2)
    in
      f3
;;
```

tests/pldi-2015-benchmarks/list_nth.ml

```
type nat =
  | O
  | S of nat

type list =
  | Nil
  | Cons of nat * list

let list_nth : list -> nat -> nat |>
  { [] => ( 0 => 0
          | 1 => 0 )
  | [2] => ( 0 => 2
```

```
          | 1 => 0 )
  | [1; 2] => ( 0 => 1
              | 1 => 2 )
  | [1] => ( 0 => 1
           | 1 => 0 )
  | [2; 1] => ( 0 => 2
              | 1 => 1 )
  | [3; 2; 1] => ( 0 => 3
                 | 1 => 2
                 | 2 => 1 )
  } = ?
```

```
let list_nth : list -> nat -> nat =
  let rec f1 (l1:list) : nat -> nat =
    fun (n1:nat) ->
      match n1 with
        | O -> (match l1 with
                  | Nil -> O
                  | Cons (n2, l2) -> n2)
        | S (n2) -> (match l1 with
                       | Nil -> O
                       | Cons (n3, l2) -> f1 l2 n2)
  in
    f1
;;
```

tests/pldi-2015-benchmarks/list_pairwise_swap.ml

```
type nat =
  | O
  | S of nat

type list =
  | Nil
  | Cons of nat * list

let list_pairwise_swap : list -> list |>
{ [] => []
| [0] => []
| [1] => []
| [0;1] => [1;0]
| [1;0] => [0;1]
| [1;0;1] => []
| [0;1;0;1] => [1;0;1;0]
} = ?
(*
{ [] => []
| [0] => []
| [1] => []
```

```
| [2] => []
| [2;2] => [2;2]
| [0;1] => [1;0]
| [1;0] => [0;1]
| [1;2] => [2;1]
| [2;1] => [1;2]
| [0;2] => [2;0]
| [0;1;0] => []
| [0;1;0;1] => [1;0;1;0]
| [1;0;1;0] => [0;1;0;1]
| [1;2;1;2] => [2;1;2;1]
| [2;1;2;1] => [1;2;1;2]
| [0;2;0;2] => [2;0;2;0]
} = ?
*)
```

```
let list_pairwise_swap : list -> list =
  let rec f1 (l1:list) : list =
    match l1 with
      | Nil -> Nil
      | Cons (n1, l2) -> (match f1 l2 with
                          | Nil -> (match l2 with
                                    | Nil -> Nil
                                    | Cons (n2, l3) -> Cons (n2,
                                                             Cons (n1, f1 l3)))
                          | Cons (n2, l3) -> Nil)
  in
    f1
;;
```

---

tests/pldi-2015-benchmarks/list_rev_append.ml

```
type nat =
  | O
  | S of nat

type list =
  | Nil
  | Cons of nat * list

let rec append (l1:list) (l2:list) : list =
  match l1 with
  | Nil -> l2
  | Cons (x, l1) -> Cons (x, append l1 l2)
;;

let list_rev_append : list -> list |>
  { [] => []
  | [0] => [0]
```

```
  | [1] => [1]
  | [0;1] => [1;0]
  | [0;0;1] => [1;0;0]
  } = ?
```

```
let list_rev_append : list -> list =
  let rec f1 (l1:list) : list =
    match l1 with
      | Nil -> Nil
      | Cons (n1, l2) -> append (f1 l2) (Cons (n1, Nil))
  in
    f1
;;
```

tests/pldi-2015-benchmarks/list_rev_fold.ml

```
type nat =
  | O
  | S of nat

type list =
  | Nil
  | Cons of nat * list

let rec fold (l:list) (f:list -> nat -> list) (acc:list) : list =
  match l with
  | Nil -> acc
  | Cons (x, l) -> fold l f (f acc x)
;;


let snoc : list -> nat -> list =
  let rec f (l:list) : nat -> list =
    fun (n:nat) ->
      match l with
      | Nil -> Cons (n, Nil)
      | Cons (x, xs) -> Cons (x, f xs n)
  in
    f
;;

let list_rev_fold : list -> list |>
  { [] => []
  | [0] => [0]
  | [1] => [1]
  | [0;1] => [1;0]
  | [0;0;1] => [1;0;0]
  } = ?
```

```
let list_rev_fold : list -> list =
  fun (l1:list) ->
    fold l1 (fun (l1:list) -> fun (n1:nat) -> Cons (n1, l1)) Nil
;;
```

tests/pldi-2015-benchmarks/list_rev_snoc.ml

```
type nat =
  | O
  | S of nat

type list =
  | Nil
  | Cons of nat * list

let snoc : list -> nat -> list =
  let rec f (l:list) : nat -> list =
    fun (n:nat) ->
      match l with
      | Nil -> Cons (n, Nil)
      | Cons (x, xs) -> Cons (x, f xs n)
  in
    f
;;

let list_rev_snoc : list -> list |>
  { [] => []
  | [0] => [0]
  | [1] => [1]
  | [0;1] => [1;0]
  | [0;0;1] => [1;0;0]
  } = ?
```

```
let list_rev_snoc : list -> list =
  let rec f1 (l1:list) : list =
    match l1 with
      | Nil -> Nil
      | Cons (n1, l2) -> snoc (f1 l2) n1
  in
    f1
;;
```

tests/pldi-2015-benchmarks/list_rev_tailcall.ml

```
type nat =
  | O
```

```
  | S of nat

type list =
  | Nil
  | Cons of nat * list

let list_rev_tailcall : list -> list -> list |>
  { [] => ( [] => []
          | [0] => [0]
          | [1] => [1]
          | [1;0] => [1;0]
          )
  | [0] => ( [] => [0] )
  | [1] => ( [] => [1]
           | [0] => [1;0]
           )
  | [0;1] => ( [] => [1;0] )
  } = ?
```

```
let list_rev_tailcall : list -> list -> list =
  let rec f1 (l1:list) : list -> list =
    fun (l2:list) ->
      match l1 with
        | Nil -> l2
        | Cons (n1, l3) -> f1 l3 (Cons (n1, l2))
  in
    f1
;;
```

tests/pldi-2015-benchmarks/list_snoc.ml

```
type nat =
  | O
  | S of nat

type list =
  | Nil
  | Cons of nat * list

let list_snoc : list -> nat -> list |>
  { [] => ( 0 => [0]
          | 1 => [1] )
  | [0] => ( 0 => [0; 0]
           | 1 => [0; 1] )
  | [1; 0] => ( 0 => [1; 0; 0]
              | 1 => [1; 0; 1] )
  | [2; 1; 0] => ( 0 => [2; 1; 0; 0]
                 | 1 => [2; 1; 0; 1] )
  } = ?
```

```
let list_snoc : list -> nat -> list =
  let rec f1 (l1:list) : nat -> list =
    fun (n1:nat) ->
      match l1 with
        | Nil -> Cons (n1, Nil)
        | Cons (n2, l2) -> Cons (n2, f1 l2 n1)
  in
    f1
;;
```

tests/pldi-2015-benchmarks/list_sort_sorted_insert.ml

```
type nat =
  | O
  | S of nat

type list =
  | Nil
  | Cons of nat * list

type cmp =
  | LT
  | EQ
  | GT

let rec compare (n1 : nat) (n2 :nat) : cmp =
  match n1 with
  | O -> ( match n2 with
           | O -> EQ
           | S (m) -> LT
         )
  | S (m1) ->
      ( match n2 with
      | O -> GT
      | S (m2) -> (compare m1 m2) )
;;

let rec insert (l : list) (n :nat) : list =
  match l with
  | Nil -> Cons(n, Nil)
  | Cons(m, tl) ->
    (match compare n m with
     | LT -> Cons (n, Cons(m, tl))
     | EQ -> l
     | GT -> Cons (m, insert tl n)
    )
;;

let list_sort_sorted_insert : list -> list |>
```

```
 { [] => []
 | [0] => [0]
 | [1] => [1]
 | [0;0] => [0]
 | [1;0] => [0;1]
 | [1;1] => [1]
 | [0;1;1] => [0;1]
 } = ?
```

```
let list_sort_sorted_insert : list -> list =
  let rec f1 (l1:list) : list =
    match l1 with
      | Nil -> Nil
      | Cons (n1, l2) -> insert (f1 l2) n1
  in
    f1
;;
```

tests/pldi-2015-benchmarks/list_sorted_insert.ml

```
type nat =
  | O
  | S of nat

type list =
  | Nil
  | Cons of nat * list

type cmp =
  | LT
  | EQ
  | GT

let rec compare (n1 : nat) (n2 :nat) : cmp =
  match n1 with
  | O -> ( match n2 with
           | O -> EQ
           | S (m) -> LT
         )
  | S (m1) ->
      ( match n2 with
      | O -> GT
      | S (m2) -> (compare m1 m2) )
;;

let list_sorted_insert : list -> nat -> list |>
  { [] => ( 0 => [0]
          | 1 => [1]
          | 2 => [2] )
```

```
  | [0] => ( 0 => [0]
           | 1 => [0;1] )
  | [1] => ( 0 => [0;1]
           | 1 => [1]
           | 2 => [1;2] )
  | [2] => ( 0 => [0;2]
           | 1 => [1;2])
  | [0;1] => ( 0 => [0;1]
             | 2 => [0;1;2] )
  } = ?
```

```
let list_sorted_insert : list -> nat -> list =
  let rec f1 (l1:list) : nat -> list =
    fun (n1:nat) ->
      match l1 with
        | Nil -> Cons (n1, Nil)
        | Cons (n2, l2) -> (match compare n2 n1 with
                            | LT -> Cons (n2, f1 l2 n1)
                            | EQ -> l1
                            | GT -> Cons (n1, l1))
  in
    f1
;;
```

tests/pldi-2015-benchmarks/list_stutter.ml

```
type nat =
  | O
  | S of nat

type list =
  | Nil
  | Cons of nat * list

let list_stutter : list -> list |>
  { [] => []
  | [0] => [0;0]
  | [1;0] => [1;1;0;0]
  } = ?
```

```
let list_stutter : list -> list =
  let rec f1 (l1:list) : list =
    match l1 with
      | Nil -> Nil
      | Cons (n1, l2) -> Cons (n1, Cons (n1, f1 l2))
  in
    f1
;;
```

tests/pldi-2015-benchmarks/list_sum.ml

```
type nat =
  | O
  | S of nat

type list =
  | Nil
  | Cons of nat * list

let rec fold (l:list) (f:nat -> nat -> nat) (acc:nat) : nat =
  match l with
  | Nil -> acc
  | Cons (x, l) -> fold l f (f acc x)
;;

let rec add (n1:nat) (n2:nat) : nat =
  match n1 with
  | O -> n2
  | S (n1) -> S (add n1 n2)
;;

let list_sum : list -> nat |>
  { [] => 0
  | [1] => 1
  | [2; 1] => 3
  } = ?
```

```
let list_sum : list -> nat =
  fun (l1:list) -> fold l1 add O
;;
```

tests/pldi-2015-benchmarks/list_take.ml

```
type nat =
  | O
  | S of nat

type list =
  | Nil
  | Cons of nat * list

let list_take : nat -> list -> list |>
  { 0 => ( [] => []
         | [1] => []
         | [0;1] => []
         | [1;0;1] => [] )
  | 1 => ( []    => []
```

```
          | [1]   => [1]
          | [0;1] => [0]
          | [1;0;1] => [1] )
  | 2 => ( []    => []
          | [1]   => [1]
          | [0;1] => [0;1]
          | [1;0;1] => [1;0] )
  } = ?
```

```
let list_take : nat -> list -> list =
  let rec f1 (n1:nat) : list -> list =
    fun (l1:list) ->
      match n1 with
        | O -> Nil
        | S (n2) -> (match l1 with
                       | Nil -> Nil
                       | Cons (n3, l2) -> Cons (n3, f1 n2 l2))
  in
    f1
;;
```

tests/pldi-2015-benchmarks/list_tl.ml

```
type nat =
  | O
  | S of nat

type list =
  | Nil
  | Cons of nat * list

let list_tl : list -> list |>
  { [] => []
  | [0] => []
  | [0; 0] => [0] } = ?
```

```
let list_tl : list -> list =
  fun (l1:list) -> match l1 with
                     | Nil -> Nil
                     | Cons (n1, l2) -> l2
;;
```

tests/pldi-2015-benchmarks/nat_iseven.ml

```
#use "nat.decls"
```

```
let nat_iseven : nat -> bool |>
  { 0 => True
  | 1 => False
  | 2 => True
  | 3 => False
  } = ?
```

```
let nat_iseven : nat -> bool =
  let rec f1 (n1:nat) : bool =
    match n1 with
      | O -> True
      | S (n2) -> (match n2 with
                     | O -> False
                     | S (n3) -> f1 n3)
  in
    f1
;;
```

tests/pldi-2015-benchmarks/nat_max.ml

```
#use "nat.decls"
#use "compare.decls"

let nat_max : nat -> nat -> nat |>
{
  0 => ( 0 => 0
       | 1 => 1
       | 2 => 2 )
| 1 => ( 0 => 1
       | 1 => 1
       | 2 => 2 )
| 2 => ( 0 => 2
       | 1 => 2
       | 2 => 2 )
} = ?
```

```
let nat_max : nat -> nat -> nat =
  let rec f1 (n1:nat) : nat -> nat =
    fun (n2:nat) ->
      match n1 with
        | O -> n2
        | S (n3) -> (match n2 with
                       | O -> n1
                       | S (n4) -> S (f1 n3 n4))
  in
    f1
;;
```

tests/pldi-2015-benchmarks/nat_pred.ml

```
#use "nat.decls"

let nat_pred : nat -> nat |>
  { O => O
  ; S (O) => O
  ; S (S (O)) => S (O) } = ?
```

```
let nat_pred : nat -> nat =
  fun (n1:nat) -> match n1 with
                    | O -> O
                    | S (n2) -> n2
;;
```

tests/pldi-2015-benchmarks/nat_sum.ml

```
#use "nat.decls"

let nat_add : nat -> nat -> nat |>
  { 0 => ( 0 => 0
         | 1 => 1
         | 2 => 2 )
  | 1 => ( 0 => 1
         | 1 => 2
         | 2 => 3 )
  | 2 => ( 0 => 2
         | 1 => 3
         | 2 => 4 )
  } = ?
```

```
let nat_add : nat -> nat -> nat =
  let rec f1 (n1:nat) : nat -> nat =
    fun (n2:nat) -> match n1 with
                      | O -> n2
                      | S (n3) -> S (f1 n3 n2)
  in
    f1
;;
```

tests/pldi-2015-benchmarks/tree_binsert.ml

```
type cmp =
  | CEq
  | CGt
```

```
  | CLt

type nat =
  | O
  | S of nat

type tree =
  | Leaf
  | Node of tree * nat * tree

let rec comp_nat (n1:nat) (n2:nat) : cmp =
  match n1 with
  | O -> (match n2 with
          | O -> CEq
          | S (n2) -> CLt)
  | S (n1) -> (match n2 with
               | O -> CGt
               | S (n2) -> comp_nat n1 n2)
;;

let tree_binsert : tree -> nat -> tree |>
  { Leaf => ( 0 => Node (Leaf, 0, Leaf)
            | 1 => Node (Leaf, 1, Leaf)
            | 2 => Node (Leaf, 2, Leaf))
  | Node (Leaf, 1, Leaf) => ( 0 => Node (Node (Leaf, 0, Leaf), 1, Leaf)
                            | 1 => Node (Leaf, 1, Leaf)
                            | 2 => Node (Leaf, 1, Node (Leaf, 2, Leaf)))
  | Node (Leaf, 0, Leaf) => ( 0 => Node (Leaf, 0, Leaf)
                            | 1 => Node (Leaf, 0, Node (Leaf, 1, Leaf))
                            | 2 => Node (Leaf, 0, Node (Leaf, 2, Leaf)))
  | Node (Leaf, 2, Leaf) => ( 0 => Node (Node (Leaf, 0, Leaf), 2, Leaf)
                            | 1 => Node (Node (Leaf, 1, Leaf), 2, Leaf)
                            | 2 => Node (Leaf, 2, Leaf))
  | Node (Node (Leaf, 0, Leaf), 1, Leaf) =>
      ( 0 => Node (Node (Leaf, 0, Leaf), 1, Leaf)
      | 1 => Node (Node (Leaf, 0, Leaf), 1, Leaf)
      | 2 => Node (Node (Leaf, 0, Leaf), 1, Node(Leaf, 2, Leaf)))
  | Node (Leaf, 0, Node (Leaf, 1, Leaf)) =>
      ( 2 => Node (Leaf, 0, Node (Leaf, 1, Node(Leaf, 2, Leaf))))
  | Node (Node (Leaf, 1, Leaf), 2, Leaf) =>
      ( 0 => Node (Node (Node(Leaf, 0, Leaf), 1, Leaf), 2, Leaf))
  | Node (Leaf, 1, Node (Leaf, 2, Leaf)) =>
      ( 0 => Node (Node (Leaf, 0, Leaf), 1, Node (Leaf, 2, Leaf))
      | 1 => Node (Leaf, 1, Node (Leaf, 2, Leaf)))
  | Node (Node (Leaf, 1, Leaf), 2, Leaf) =>
      ( 0 => Node (Node (Node(Leaf, 0, Leaf), 1, Leaf), 2, Leaf))
  } = ?
```

```
let tree_binsert : tree -> nat -> tree =
  let rec f1 (t1:tree) : nat -> tree =
```

```
    fun (n1:nat) ->
      match t1 with
        | Leaf -> Node (Leaf, n1, Leaf)
        | Node (t2, n2, t3) -> (match comp_nat n2 n1 with
                                  | CEq -> t1
                                  | CGt -> Node (f1 t2 n1, n2, t3)
                                  | CLt -> Node (t2, n2, f1 t3 n1))
  in
    f1
;;
```

tests/pldi-2015-benchmarks/tree_collect_leaves.ml

```
type bool =
  | True
  | False

type tree =
  | Leaf
  | Node of tree * bool * tree

type list =
  | Nil
  | Cons of bool * list

let rec append (l1:list) (l2:list) : list =
  match l1 with
  | Nil -> l2
  | Cons (x, l1) -> Cons (x, append l1 l2)
;;

let tree_collect_leaves : tree -> list |>
  { Leaf => []
  | Node (Leaf, True, Leaf) => [True]
  | Node (Leaf, False, Leaf) => [False]
  | Node (Node (Leaf, True, Leaf), False, Leaf) => [True; False]
  | Node (Node (Leaf, False, Leaf), True, Leaf) => [False; True]
  | Node (Leaf, False, Node (Leaf, True, Leaf)) => [False; True]
  } = ?
```

```
let tree_collect_leaves : tree -> list =
  let rec f1 (t1:tree) : list =
    match t1 with
      | Leaf -> Nil
      | Node (t2, b1, t3) -> append (f1 t2) (Cons (b1, f1 t3))
  in
    f1
;;
```

tests/pldi-2015-benchmarks/tree_count_leaves.ml

```
type bool =
  | True
  | False

type tree =
  | Leaf
  | Node of tree * bool * tree

type nat =
  | O
  | S of nat

let rec sum (n1:nat) (n2:nat) : nat =
  match n1 with
  | O -> n2
  | S (n1) -> S (sum n1 n2)
;;

let tree_count_leaves : tree -> nat |>
  { Leaf => 1
  | Node (Leaf, True, Leaf) => 2
  | Node (Node (Leaf, True, Leaf), True, Leaf) => 3
  | Node (Leaf, True, Node (Leaf, True, Leaf)) => 3
  | Node (Node (Node (Leaf, True, Leaf), True, Leaf), True, Leaf) => 4
  | Node (Node (Leaf, True, Leaf), True, Node (Leaf, True, Leaf)) => 4
  | Node (Node (Leaf, True, Leaf), True,
      Node (Node (Leaf, True, Leaf), True, Node (Leaf, True, Leaf))) => 6
  } = ?
```

```
let tree_count_leaves : tree -> nat =
  let rec f1 (t1:tree) : nat =
    match t1 with
      | Leaf -> S (O)
      | Node (t2, b1, t3) -> sum (f1 t2) (f1 t3)
  in
    f1
;;
```

tests/pldi-2015-benchmarks/tree_count_nodes.ml

```
type nat =
  | O
  | S of nat

type tree =
  | Leaf
```

```
  | Node of tree * nat * tree

let rec sum (n1:nat) (n2:nat) : nat =
  match n1 with
  | O -> n2
  | S (n1) -> S (sum n1 n2)
;;

let tree_count_nodes : tree -> nat |>
  { Leaf => 0
  | Node(Leaf, 0, Leaf) => 1
  | Node(Node(Leaf, 0, Leaf), 0, Leaf) => 2
  | Node(Leaf, 0, Node(Leaf, 0, Leaf)) => 2
  | Node(Node(Leaf, 0, Node(Leaf, 0, Leaf)), 0, Leaf) => 3
  | Node(Leaf, 0, Node(Leaf, 0, Node(Leaf, 0, Leaf))) => 3
  } = ?
```

```
let tree_count_nodes : tree -> nat =
  let rec f1 (t1:tree) : nat =
    match t1 with
      | Leaf -> O
      | Node (t2, n1, t3) -> S (sum (f1 t2) (f1 t3))
  in
    f1
;;
```

---

tests/pldi-2015-benchmarks/tree_inorder.ml

```
type nat =
  | O
  | S of nat

type list =
  | Nil
  | Cons of nat * list

type tree =
  | Leaf
  | Node of tree * nat * tree

let rec append (l1:list) (l2:list) : list =
  match l1 with
  | Nil -> l2
  | Cons (x, l1) -> Cons (x, append l1 l2)
;;

let tree_inorder: tree -> list |>
{ Leaf => []
| Node (Leaf, 1, Leaf) => [1]
```

```
| Node (Leaf, 2, Leaf) => [2]
| Node (Node (Leaf, 1, Leaf), 2, Leaf) => [1;2]
| Node (Leaf, 1, Node (Leaf, 2, Leaf)) => [1;2]
} = ?
```

```
let tree_inorder : tree -> list =
  let rec f1 (t1:tree) : list =
    match t1 with
      | Leaf -> Nil
      | Node (t2, n1, t3) -> append (f1 t2) (Cons (n1, f1 t3))
  in
    f1
;;
```

---

tests/pldi-2015-benchmarks/tree_map.ml

```
type nat =
  | O
  | S of nat

type list =
  | Nil
  | Cons of nat * list

type tree =
  | Leaf
  | Node of tree * nat * tree

let rec div2 (n:nat) : nat =
  match n with
  | O -> O
  | S (n1) -> match n1 with
    | O -> O
    | S (n2) -> S (div2 n2)
;;

let rec inc (n:nat) : nat =
  S( n )
;;


let tree_map : (nat -> nat) -> tree -> tree |>
{ div2 => ( Leaf => Leaf
          | Node (Leaf, 0, Leaf) => Node (Leaf, 0, Leaf)
          | Node (Leaf, 2, Leaf) => Node (Leaf, 1, Leaf)
          | Node (Node (Leaf, 2, Leaf), 2, Leaf) =>
              Node (Node (Leaf, 1, Leaf), 1, Leaf)
          | Node (Leaf, 1, Node (Leaf, 2, Leaf)) =>
              Node (Leaf, 0, Node (Leaf, 1, Leaf))
```

```
          )
| inc =>  ( Leaf => Leaf
          | Node (Leaf, 0, Leaf) => Node (Leaf, 1, Leaf) )
} = ?
```

```
let tree_map : (nat -> nat) -> tree -> tree =
  fun (f2:nat -> nat) ->
    let rec f3 (t1:tree) : tree =
      match t1 with
        | Leaf -> Leaf
        | Node (t2, n1, t3) -> Node (f3 t2, f2 n1, f3 t3)
    in
      f3
;;
```

tests/pldi-2015-benchmarks/tree_nodes_at_level.ml

```
type bool =
  | True
  | False

type tree =
  | Leaf
  | Node of tree * bool * tree

type nat =
  | O
  | S of nat

let rec sum (n1:nat) (n2:nat) : nat =
      match n1 with
      | O -> n2
      | S (n1p) -> S (sum n1p n2)
;;

let tree_nodes_at_level : tree -> nat -> nat |>
  { Leaf =>
    ( 0 => 0
    | 1 => 0
    )
  | Node (Leaf, True, Leaf) =>
    ( 0 => 1
    | 1 => 0
    )
  | Node (Node (Leaf, True, Leaf), True, Leaf) =>
    ( 0 => 1
    | 1 => 1
    )
  | Node (Node (Leaf, True, Leaf), True, Node (Leaf, True, Leaf)) =>
```

```
    ( 0 => 1
    | 1 => 2
    | 2 => 0
    )
  | Node (Node
      (Node (Leaf, True, Leaf), True, Node (Leaf, True, Leaf)), True, Leaf) =>
    ( 0 => 1
    | 1 => 1
    )
  } = ?
```

```
let tree_nodes_at_level : tree -> nat -> nat =
  let rec f1 (t1:tree) : nat -> nat =
    fun (n1:nat) ->
      match t1 with
        | Leaf -> O
        | Node (t2, b1, t3) -> (match n1 with
                                  | O -> S (O)
                                  | S (n2) -> sum (f1 t3 n2) (f1 t2 n2))
  in
    f1
;;
```

tests/pldi-2015-benchmarks/tree_postorder.ml

```
type nat =
  | O
  | S of nat

type list =
  | Nil
  | Cons of nat * list

type tree =
  | Leaf
  | Node of tree * nat * tree

let rec append (l1:list) (l2:list) : list =
  match l1 with
  | Nil -> l2
  | Cons (x, l1) -> Cons (x, append l1 l2)
;;

let tree_postorder : tree -> list |>
{  Leaf => []
| Node (Leaf, 1, Leaf) => [1]
| Node (Leaf, 2, Leaf) => [2]
| Node (Node (Leaf, 1, Leaf), 2, Leaf) => [1;2]
| Node (Leaf, 1, Node (Leaf, 2, Leaf)) => [2;1]
```

```
| Node (Node (Leaf, 1, Leaf), 0, Node (Leaf, 2, Leaf) ) => [1;2;0]
| Node (Node (Leaf, 2, Leaf), 0, Node (Leaf, 1, Leaf) ) => [2;1;0]
| Node (Node (Node (Leaf, 2, Leaf), 0, Node (Leaf, 1, Leaf) ), 0, Leaf) =>
    [2;1;0;0]
| Node (Leaf, 2, Node (Node (Leaf, 2, Leaf), 0, Node (Leaf, 1, Leaf) )) =>
    [2;1;0;2]
} = ?
```

```
let tree_postorder : tree -> list =
  let rec f1 (t1:tree) : list =
    match t1 with
      | Leaf -> Nil
      | Node (t2, n1, t3) -> (match f1 t2 with
                                | Nil -> append (f1 t3) (Cons (n1, Nil))
                                | Cons (n2, l1) -> Cons (n2,
                                                         append
                                                           (append l1 (f1 t3))
                                                           (Cons (n1, Nil))))
  in
    f1
;;
```

tests/pldi-2015-benchmarks/tree_preorder.ml

```
type nat =
  | O
  | S of nat

type list =
  | Nil
  | Cons of nat * list

type tree =
  | Leaf
  | Node of tree * nat * tree

let rec append (l1:list) (l2:list) : list =
  match l1 with
  | Nil -> l2
  | Cons (x, l1) -> Cons (x, append l1 l2)
;;

let tree_preorder : tree -> list |>
{ Leaf => []
| Node (Leaf, 1, Leaf) => [1]
| Node (Leaf, 2, Leaf) => [2]
| Node (Node (Leaf, 1, Leaf), 2, Leaf) => [2;1]
| Node (Leaf, 1, Node (Leaf, 2, Leaf)) => [1;2]
} = ?
```

```
let tree_preorder : tree -> list =
  let rec f1 (t1:tree) : list =
    match t1 with
      | Leaf -> Nil
      | Node (t2, n1, t3) -> Cons (n1, append (f1 t2) (f1 t3))
  in
    f1
;;
```

B.3 Extended Examples

tests/pldi-2015-extended/arith.ml

```
(* -matches 2 -scrutinee 8 *)

type nat =
| O
| S of nat

type cmp =
  | LT
  | EQ
  | GT

type exp =
| Const of nat
| Sum of exp * exp
| Prod of exp * exp
| Pred of exp
| Max of exp * exp

let rec sum (n1:nat) (n2:nat) : nat =
  match n1 with
  | O -> n2
  | S (n1) -> S (sum n1 n2)
;;

let rec mult (n1:nat) (n2:nat) : nat =
  match n1 with
  | O -> O
  | S (n1) -> sum n2 (mult n1 n2)
;;

let rec compare (n1 : nat) (n2 :nat) : cmp =
  match n1 with
  | O -> ( match n2 with
           | O -> EQ
```

```
          | S (m) -> LT
        )
  | S (m1) ->
      ( match n2 with
      | O -> GT
      | S (m2) -> (compare m1 m2) )
;;

let arith : exp -> nat |>
{ Const (0) => 0
| Const (1) => 1
| Const (2) => 2
| Sum (Const(2), Const(2)) => 4
| Sum (Const(2), Const(1)) => 3
| Sum (Const(0), Const(2)) => 2
| Prod (Const(0), Const(2)) => 0
| Prod (Const(2), Const(1)) => 2
| Prod (Const(2), Const(2)) => 4
| Prod (Prod(Const(2), Const(2)), Const(2)) => 8
| Prod (Sum(Const(2), Const(1)), Const(2)) => 6
| Pred (Const(0)) => 0
| Pred (Const(1)) => 0
| Pred (Const(2)) => 1
| Max (Const(0), Const(0)) => 0
| Max (Const(0), Const(1)) => 1
| Max (Const(0), Const(2)) => 2
| Max (Const(1), Const(0)) => 1
| Max (Const(1), Const(1)) => 1
| Max (Const(1), Const(2)) => 2
| Max (Const(2), Const(0)) => 2
| Max (Const(2), Const(1)) => 2
} = ?
```

```
let arith : exp -> nat =
  let rec f1 (e1:exp) : nat =
    match e1 with
      | Const (n1) -> n1
      | Sum (e2, e3) -> sum (f1 e2) (f1 e3)
      | Prod (e2, e3) -> mult (f1 e2) (f1 e3)
      | Pred (e2) -> (match f1 e2 with
                        | O -> O
                        | S (n1) -> n1)
      | Max (e2, e3) -> (match compare (f1 e2) (f1 e3) with
                           | LT -> f1 e3
                           | EQ -> f1 e3
                           | GT -> f1 e2)
  in
    f1
;;
```

---

tests/pldi-2015-extended/dyn_app_twice.ml

```
type nat =
   | O
   | S of nat

type dyn =
  | Error
  | Base of nat
  | Dyn of (dyn -> dyn)


let succ (d:dyn) : dyn =
  match d with
  | Error -> Error
  | Base ( n ) -> Base (S(n))
  | Dyn ( f ) -> Error
;;

let pred (d:dyn) : dyn =
  match d with
  | Error -> Error
  | Base ( n ) ->
    (match n with
    | O -> Base ( O )
    | S ( n ) -> Base ( n ))
  | Dyn ( f ) -> Error
;;

let dyn_app_twice : dyn -> dyn -> dyn |>
{
  Dyn (succ) => ( Base( 0 ) => Base( 2 )
                | Base( 1 ) => Base( 3 ) )
| Dyn (pred) => ( Base( 0 ) => Base( 0 )
                | Base( 1 ) => Base( 0 ) )
| Error => Error => Error
| Base( 0 ) => Error => Error
} = ?
```

```
let dyn_app_twice : dyn -> dyn -> dyn =
  fun (d1:dyn) ->
    fun (d2:dyn) ->
      match d1 with
        | Error -> Error
        | Base (n1) -> Error
        | Dyn (f3) -> f3 (f3 d2)
;;
```

tests/pldi-2015-extended/dyn_sum.ml

```
type nat =
   | O
   | S of nat

type dyn =
  | Error
  | Base of nat
  | Dyn of (dyn -> dyn)


let succ (d:dyn) : dyn =
  match d with
  | Error -> Error
  | Base ( n ) -> Base (S(n))
  | Dyn ( f ) -> Error
;;

let id (d:dyn) : dyn = d ;;

let dyn_sum : dyn -> dyn -> dyn |>
{ Dyn (id) =>
  ( Error    => Error
  | Dyn (id) => Error
  | Base (0) => Error
  | Base (1) => Error
  | Base (2) => Error )
| Error =>
  ( Error    => Error
  | Dyn (id) => Error
  | Base (0) => Error
  | Base (1) => Error
  | Base (2) => Error )
| Base (0) =>
  ( Error    => Error
  | Dyn (id) => Error
  | Base (0) => Base (0)
  | Base (1) => Base (1)
  | Base (2) => Base (2) )
| Base (1) =>
  ( Error    => Error
  | Dyn (id) => Error
  | Base (0) => Base (1)
  | Base (1) => Base (2)
  | Base (2) => Base (3) )
| Base (2) =>
  ( Error    => Error
  | Dyn (id) => Error
  | Base (0) => Base (2)
  | Base (1) => Base (3)
  | Base (2) => Base (4) )
```

```
} = ?
```

```
let dyn_sum : dyn -> dyn -> dyn =
  fun (d1:dyn) ->
    fun (d2:dyn) ->
      match d1 with
        | Error -> Error
        | Base (n1) -> (match n1 with
                          | O -> (match d2 with
                                    | Error -> Error
                                    | Base (n2) -> d2
                                    | Dyn (f3) -> Error)
                          | S (n2) -> (match n2 with
                                         | O -> succ d2
                                         | S (n3) -> succ (succ d2)))
        | Dyn (f3) -> Error
;;
```

tests/pldi-2015-extended/fvs_large.ml

```
type nat =
  | O
  | S of nat

type list =
  | Nil
  | Cons of nat * list

type exp =
  | Unit
  | BVar of nat
  | FVar of nat
  | Lam of nat * exp
  | App of exp * exp
  | Pair of exp * exp
  | Fst of exp
  | Snd of exp
  | Inl of exp
  | Inr of exp
  | Const of nat
  | Add of exp * exp
  | Sub of exp * exp
  | Mult of exp * exp
  | Div of exp * exp

let rec append (l1:list) (l2:list) : list =
  match l1 with
  | Nil -> l2
  | Cons (x, l1p) -> Cons (x, append l1p l2)
```

```
;;

let fvs_large : exp -> list |>
  { Unit => []
  | FVar (0) => [0]
  | FVar (1) => [1]
  | FVar (2) => [2]
  | BVar (0) => []
  | Lam (0, Unit) => []
  | Lam (0, FVar (1)) => [1]
  | App (Unit, Unit) => []
  | App (FVar (0), Unit) => [0]
  | App (Unit, FVar (1)) => [1]
  | Fst (Unit) => []
  | Fst (FVar (1)) => [1]
  | Snd (Unit) => []
  | Snd (FVar (1)) => [1]
  | Pair (Unit, Unit) => []
  | Pair (FVar (0), Unit) => [0]
  | Pair (Unit, FVar (1)) => [1]
  | Pair (FVar (0), FVar (1)) => [0; 1]
  | Inl (Unit) => []
  | Inl (FVar (1)) => [1]
  | Inr (Unit) => []
  | Inr (FVar (1)) => [1]
  | Const (0) => []
  | Add (FVar (0), Unit) => [0]
  | Add (Unit, FVar (1)) => [1]
  | Sub (FVar (0), Unit) => [0]
  | Sub (Unit, FVar (1)) => [1]
  | Mult (FVar (0), Unit) => [0]
  | Mult (Unit, FVar (1)) => [1]
  | Div (FVar (0), Unit) => [0]
  | Div (Unit, FVar (1)) => [1]
  } = ?
```

```
let fvs_large : exp -> list =
  let rec f1 (e1:exp) : list =
    match e1 with
      | Unit -> []
      | BVar (n1) -> []
      | FVar (n1) -> [n1]
      | Lam (n1, e2) -> f1 e2
      | App (e2, e3) -> append (f1 e2) (f1 e3)
      | Pair (e2, e3) -> append (f1 e2) (f1 e3)
      | Fst (e2) -> f1 e2
      | Snd (e2) -> f1 e2
      | Inl (e2) -> f1 e2
      | Inr (e2) -> f1 e2
      | Const (n1) -> []
```

```
      | Add (e2, e3) -> append (f1 e2) (f1 e3)
      | Sub (e2, e3) -> append (f1 e2) (f1 e3)
      | Mult (e2, e3) -> append (f1 e2) (f1 e3)
      | Div (e2, e3) -> append (f1 e2) (f1 e3)
  in
    f1
;;
```

---

tests/pldi-2015-extended/fvs_medium.ml

```
type nat =
  | O
  | S of nat

type list =
  | Nil
  | Cons of nat * list

type binop =
  | Add
  | Sub
  | Mul
  | Div

type exp =
  | Unit
  | BVar of nat
  | FVar of nat
  | Lam of nat * exp
  | App of exp * exp
  | Pair of exp * exp
  | Fst of exp
  | Snd of exp
  | Const of nat
  | Binop of exp * binop * exp

let rec append (l1:list) (l2:list) : list =
  match l1 with
  | Nil -> l2
  | Cons (x, l1p) -> Cons (x, append l1p l2)
;;

let fvs_medium : exp -> list |>
  { Unit => []
  | BVar (0) => [0]
  | BVar (1) => [1]
  | BVar (2) => [2]
  | FVar (0) => []
  | Lam (0, Unit) => []
  | Lam (0, BVar (1)) => [1]
```

```
  | App (Unit, Unit) => []
  | App (BVar (0), Unit) => [0]
  | App (Unit, BVar (1)) => [1]
  | App (BVar (0), BVar (1)) => [0; 1]
  | Fst (Unit) => []
  | Fst (BVar (1)) => [1]
  | Snd (Unit) => []
  | Snd (BVar (1)) => [1]
  | Pair (Unit, Unit) => []
  | Pair (BVar (0), Unit) => [0]
  | Pair (Unit, BVar (1)) => [1]
  | Pair (BVar (0), BVar (1)) => [0; 1]
  | Const (0) => []
  | Binop (BVar (0), Add, Unit) => [0]
  | Binop (Unit, Add, BVar (1)) => [1]
  } = ?
```

```
let fvs_medium : exp -> list =
  let rec f1 (e1:exp) : list =
    match e1 with
      | Unit -> Nil
      | BVar (n1) -> Cons (n1, Nil)
      | FVar (n1) -> Nil
      | Lam (n1, e2) -> f1 e2
      | App (e2, e3) -> append (f1 e2) (f1 e3)
      | Pair (e2, e3) -> append (f1 e2) (f1 e3)
      | Fst (e2) -> f1 e2
      | Snd (e2) -> f1 e2
      | Const (n1) -> Nil
      | Binop (e2, b1, e3) -> append (f1 e2) (f1 e3)
  in
    f1
;;
```

tests/pldi-2015-extended/fvs_small.ml

```
type id =
  | A
  | B
  | C

type nat =
  | O
  | S of nat

type list =
  | Nil
  | Cons of id * list
```

```
type exp =
  | EBVar of nat
  | EFVar of id
  | EApp  of exp * exp

let rec append (l1:list) (l2:list) : list =
  match l1 with
  | Nil -> l2
  | Cons (x, l1p) -> Cons (x, append l1p l2)
;;

let fvs_small : exp -> list |>
  { EBVar (0) => []
  | EFVar (A) => [A]
  | EFVar (B) => [B]
  | EApp (EBVar (0), EBVar (0)) => []
  | EApp (EFVar (A), EBVar (0)) => [A]
  | EApp (EBVar (0), EFVar (A)) => [A]
  } = ?
```

```
let fvs_small : exp -> list =
  let rec f1 (e1:exp) : list =
    match e1 with
      | EBVar (n1) -> Nil
      | EFVar (i1) -> Cons (i1, Nil)
      | EApp (e2, e3) -> append (f1 e2) (f1 e3)
  in
    f1
;;
```

Bibliography

Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. Recursive program synthesis. In *Proceedings of the 25th Conference on Computer-Aided Verification*, 2013.

Rajeev Alur, Pavol Černý, P. Madhusudan, and Wonhong Nam. Synthesis of interface specifications for java classes. In *Proceedings of the 32nd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2005.

Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishk Udupa. Syntax-guided synthesis. In *13th International Conference on Formal Methods in Computer-aided Design*, 2013.

Alan Ross Anderson, Nuel D. Belnap, and J. Michael Dunn. *Entailment: The logic of relevance and necessity, vol. II*. Princeton University Press, Princeton, 1992.

Lennart Augustsson. [haskell] announcing djinn, version 2004-12-11, a coding wizard. Mailing List, 2004. `http://www.haskell.org/pipermail/haskell/2005-December/017055.html`.

Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2008.

Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. *Satisfiability Modulo Theories*, chapter 12, pages 737–797. IOS Press, 2008.

Rastislav Bodik, Satish Chandra, Joel Galenson, Doug Kimelman, Nicholas Tung, Shaon Barman, and Casey Rodarmor. Programming with angelic nondeterminism. In *Proceedings of the 37th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2010.

Forrest Briggs and Melissa O'Neill. Functional genetic programming and exhaustive program search with combinator expressions. *International Journal*

of Knowledge-based and Intelligent Engineering Systems–Genetic Programming An Emerging Engineering Tool, 12(1), 2008.

Koen Claessen, Jonas Duregård, and Michał H. Pałka. Generating constrained random data with uniform distribution. In *Functional and Logic Programming*. Springer International Publishing, 2014.

Leonardo De Moura and Nikolaj Björner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

John K. Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2015.

Pierre Flener and Serap Yilmaz. Inductive synthesis of recursive logic programs: Achievements and prospects. *The Journal of Logic Programming*, 41(23):141 – 195, 1999.

Jonathan Frankle. Type-directed synthesis of products. Master's thesis, Princeton University, 2015.

Tim Freeman and Frank Pfenning. Refinement types for ml. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation (PLDI)*, 1991.

Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C. Pierce. Linear dependent types for differential privacy. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL)*, 2013.

Jean-Yves Girard. *Interpèation Fonctionnelle et Èlimination des Coupures de L'arithmètique D'ordre Supèrieur.* PhD thesis, Universitè Paris VII, 1972.

Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

Cordell Green. Application of theorem proving to problem solving. In *International Joint Conference on Artificial Intelligence*, 1969.

Katarzyna Grygiel and Pierre Lescanne. Counting and generating lambda terms. *Journal of Functional Programming*, 23:594–628, 9 2013. ISSN 1469-7653.

Sumit Gulwani. Dimensions in program synthesis. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP)*, 2010.

Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2011.

Sumit Gulwani and Nebojsa Jojic. Program verification as probabilistic inference. In *Proceedings of the 34th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2007.

Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. Complete completion using types and weights. In *Proceedings of the 2013 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2013.

Martin Hofmann, Emanuel Kitzelmann, and Ute Schmid. Porting IGOR II from Maude to Haskell. In Ute Schmid, Emanuel Kitzelmann, and Rinus Plasmeijer, editors, *Approaches and Applications of Inductive Programming*, volume 5812 of *Lecture Notes in Computer Science*, pages 140–158. Springer Berlin Heidelberg, 2010.

William A. Howard. The formulae-as-types notion of construction. In Jonathan P. Seldin and J. Roger Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. Academic Press, 1980.

Paul Hudak. Domain specific languages. In Peter H. Salas, editor, *Handbook of Programming Languages*, chapter 3, pages 39–60. MacMillan, 1998.

Limin Jia, Jeffrey A. Vaughan, Karl Mazurak, Jianzhou Zhao, Luke Zarko, Joseph Schorr, and Steve Zdancewic. AURA: A programming language for authorization and audit. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, 2008.

Stefan Kahrs. The primitive recursive functions are recursively enumerable. http://www.cs.kent.ac.uk/people/staff/smk/primrec.pdf.

Susumu Katayama. An analytical inductive functional programming system that avoids unintended programs. In *Proceedings of the ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation*, PEPM '12, pages 43–52, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1118-2.

Emanuel Kitzelmann. Inductive programming: A survey of program synthesis techniques. In Ute Schmid, Emanuel Kitzelmann, and Rinus Plasmeijer, editors, *Approaches and Applications of Inductive Programming*, volume 5812 of *Lecture Notes in Computer Science*, pages 50–73. Springer Berlin Heidelberg, 2010a. ISBN 978-3-642-11930-9.

Emanuel Kitzelmann. *A Combined Analytical and Search-based Approach to the Inductive Synthesis of Functional Programs*. PhD thesis, Fakulät für Wirtschafts-und Angewandte Informatik, Universität Bamberg, 2010b.

Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. Synthesis modulo recursive functions. In *Proceedings of the 28th ACM SIGPLAN on Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2013.

Christoph Kreitz. *Program Synthesis*, pages 105–134. Springer, 1998.

Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. Complete functional synthesis. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2010.

Tessa Lau. *Programming by Demonstration: a Machine Learning Approach*. PhD thesis, University of Washington, 2001.

Vu Le and Sumit Gulwani. Flashextract: A framework for data extraction by examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14. ACM, 2014.

Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rèmy, and Jèrôme Vouillon. *The OCaml System Release 4.02: Documentation and User's Manual*. http://caml.inria.fr/pub/docs/manual-ocaml/, 2014.

Chuck Liang and Dale Miller. Focusing and polarization in intuitionistic logic. In Jacques Duparc and ThomasA. Henzinger, editors, *Computer Science Logic*, Lecture Notes in Computer Science, pages 451–465. Springer Berlin Heidelberg, 2007.

David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: Helping to navigate the api jungle. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

Zohar Manna and Richard Waldinger. Synthesis: Dreams $\rightarrow$ programs. *IEEE Transactions on Software Engineering*, 4, July 1979.

Zohar Manna and Richard Waldinger. A deductive approach to program synthesis. *ACM Trans. Program. Lang. Syst.*, 2(1):90–121, January 1980. ISSN 0164-0925.

Per Martin-Löf. *Intuitionistic type theory*. Bibliopolis, 1984.

Karl Mazurak and Steve Zdancewic. Lolliproc: to concurrency from classical linear logic via curry-howard and control. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, 2010.

Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), 1965.

Stephen Muggleton and Luc de Raedt. Inductive logic programming: Theory and methods. *The Journal of Logic Programming*, 19(20):629–679, 1994.

Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM*, 27, 1980.

Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. In *Proceedings of the 2015 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2015.

Daniel Perelman, Sumit Gulwani, Thomas Ball, and Dan Grossman. Type-directed completion of partial expressions. In *Proceedings of the 2012 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2012.

Daniel Perelman, Sumit Gulwani, Dan Grossman, and Peter Provost. Test-driven synthesis. In *Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for gadts. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2006.

Simon Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003. `http://www.haskell.org/definition/`.

Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, January 2000. ISSN 0164-0925.

Dimitrios Prountzos, Roman Manevich, and Keshav Pingali. Elixir: A system for synthesizing concurrent graph programs. In *Proceedings of the 27th ACM SIGPLAN on Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2012.

Alexey Rodriguez Yakushev and Johan Jeuring. Enumerating well-typed terms generically. In *Approaches and Applications of Inductive Programming*. Springer Berlin Heidelberg, 2010.

Sriram Sankaranarayanan, Franjo Ivančić, and Aarti Gupta. Mining library specifications using inductive logic programming. In *Proceedings of the 30th International Conference On Software Engineering*, 2008.

Gabriel Scherer and Didier Rèmy. Which simple types have a unique inhabitant? In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2015.

Rohah Shah. Type-directed program synthesis with record types. Master's thesis, University of Pennsylvania, 2015.

Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. In *Proceedings of the 2013 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2013.

Armando Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, University of California, Berkeley, 2008.

Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodik. Sketching concurrent data structures. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2008.

Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. From program verification to program synthesis. In *Proceedings of the 37th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2010.

Phillip D. Summers. A methodology for LISP program construction from examples. In *Proceedings of the 3rd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 1976.

Don Syme. *The F# 3.1 Language Specification*. http://fsharp.org/specs/language-spec/3.1/FSharpSpec-3.1-working.docx, 2013.

W. W. Tait. Intensional interpretations of functionals of finite type I. *The Journal of Symbolic Logic*, 32(2):198–212, 1967.

The Coq Development Team. *The Coq Proof Assistant: Reference Manual*. INRIA, http://coq.inria.fr/distrib/current/refman/, 2012.

Emina Torlak and Rastislav Bodik. A lightweight symbolic virtual machine for solver-aided host languages. In *Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014.

Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M.K. Martin, and Rajeev Alur. Transit: Specifying protocols with concolic snippets. In *Proceedings of the 2013 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2013.

Philip Wadler. Is there a use for linear logic? In *Proceedings of the 1991 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM)*, 1991.

David Walker. Substructural type systems. In *Advanced Types and Programming Languages*, chapter 1, pages 3–34. The MIT Press, 2005.

Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, 2009.

Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2003.

Pavol Černý, Krishnendu Chatterjee, ThomasA. Henzinger, Arjun Radhakrishna, and Rohit Singh. Quantitative synthesis for concurrent programs. In *Proceedings of the 23rd Conference on Computer-Aided Verification*, 2011.