Programming "Programming by Example" by Example

A THESIS PRESENTED
BY
GARRETT TANZER
TO
THE DEPARTMENT OF COMPUTER SCIENCE

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
BACHELOR OF ARTS
IN THE SUBJECT OF
COMPUTER SCIENCE

HARVARD UNIVERSITY
CAMBRIDGE, MASSACHUSETTS
MAY 2020

Advisor: Nada Amin Garrett Tanzer

Programming "Programming by Example" by Example

Abstract

Programming by Example (PbE) is an increasingly popular model for writing programs to automate repetitive tasks, such as those common in data wrangling. In this paradigm, the user presents a set of *input/output pairs*, and an *inductive synthesizer* responds with a program that satisfies these examples. However, due to performance limitations, these synthesizers must be specialized to the domain at hand. Frameworks like PROSE [10] have been developed to ease the implementation burden when writing synthesizers for domain-specific languages (DSLs), but they remain relatively inaccessible to those who are not experts in programming languages and synthesis. We identify three key challenges facing the users of such tools: designing DSLs, writing deduction rules, and debugging.

In order to remove these barriers, we present a new interaction model for developing inductive program synthesizers *by example*. Rather than write an explicit DSL grammar, the developer communicates their intended domain implicitly, by providing a corpus of programs in a general-purpose language. From these examples, a compiler infers an embedded DSL and outputs an inductive synthesizer for this specialized domain.

We demonstrate the viability of this approach with VERSE (VERified Synthesis Engine), a Coq implementation of the *Programming "Programming by Example" by Example* (PbE^2) interaction model. VERSE comprises a toy general-purpose language and its library of operators, the aforementioned DSL compiler, and a synthesis engine that uses top-down deductive search à la PROSE. It is architected to enable an end-to-end, mechanically verified proof of correctness—this component is under ongoing development.

VERSE's implementation is publicly available at https://github.com/gtanzer/verse.

Contents

Acknowledgments

I would like to thank my wonderful advisors Nada Amin and Eddie Kohler for guiding me along this journey, as well as for introducing me to program synthesis and Coq respectively. This thesis would look quite different were it not for both of them. My other professors—Steve Chong, Boaz Barak, Stratos Idreos, James Mickens, Michael Mitzenmacher, and many others—were likewise instrumental in this process, by helping to mold me into the computer scientist I am today. And a special shout-out to Elena Glassman for co-teaching (with Nada) the class in which this project was born!

I couldn't have done any of this without the support of my family and friends, who kept me (relatively) sane through all of the twists and turns, of course culminating in quarantine. More literally, I perhaps could have done it without Michael Colavita and Luke Melas-Kyriazi, but the resulting paper would have had more typos and uncomfortable phrasings; I am thankful for their proofreading.

I would also like to thank David Holland for his many Coq tips over the years, as well as more recently Samuel Gruetter and the denizens of the #coq Freenode IRC channel for their assistance on some arcane errors I experienced while implementing this paper's artifact.

Parts of this paper—specifically the abstract, introduction, related work, and design sections—draw from previous work conducted jointly with Alex Wendland [79]. This prior work focuses on the usability of the PbE^2 interaction model evaluated through user studies, whereas the current work provides a formal treatment of PbE^2 and its implementation in VERSE.

1. Introduction

Many users face tasks that could be automated, but they don't know how to write programs to do so [54]. For example, a user may be given a spreadsheet with a column containing the entries [36]:

Alice Smith, 1956-05-30
Bob H. Johnson, 1996-12-20
Charlie Williams, 1984-11-23
...

If the user wants to extract names into one column with the format "Lastname, Firstname Middleinitial", and dates into another with the format "mm-dd-yy", they may resort to doing this manually. Even if the user is a competent programmer, they may decide that writing code is not worth the time investment, especially for relatively short tasks [11].

Programming by Example (PbE) is an interaction model for programming in which the user communicates their intent by providing *input/output pairs* as a form of specification, rather than by writing code. An application called an *inductive synthesizer* responds by constructing a program that computes these mappings, and which will hopefully generalize to the function that the user has in mind. Often this process is interactive, where the user exchanges new examples to clarify their intent until the synthesizer outputs a program with the desired functionality [43]. Returning to our spreadsheet example, the user might specify the format of the name column by telling the synthesizer that the input "Alice Smith, 1956-05-30" should map to the output "Smith, Alice". The synthesizer responds that it doesn't know how to handle cases like the second row, which could be formatted in several ways, like "H. Bob", "H. Johnson, Bob", "Johnson, Bob H.", etc. The user clarifies their intent by adding an additional example, "Bob H. Johnson, 1996-12-20" $\rightarrow$ "Johnson, Bob H.". Given this less ambiguous specification of two examples, the synthesizer generates a program to compute the transformation and applies it to the entire input column, sparing the user a lot of tedium.

Because this interaction model requires no programming, it can be easily integrated into GUI applications, enabling this substantial fraction of users who lack access to traditional programming methods to automate parts of their workflows, using an interface that they are already familiar with. Even for users who do know how to code, specifying intent by example is often faster than writing a program manually, expanding the class of tasks that can feasibly be automated. Text transformations and data wrangling, as alluded to above, are the canonical examples of these kinds of tasks; data scientists are often confronted with idiosyncratic unstructured data, which must be parsed and transformed into a more useful format. Microsoft's FlashFill [35] and FlashExtract [44] are technologies that target precisely this use case, help-

ing users transform the content of columns in Excel and extract information from unstructured text files respectively.

Note that these domains are extremely restricted, compared to programming as a whole. Rather than output programs in a general-purpose language (GPL) like C++ or Python, most existing synthesizers work within a domain-specific language (DSL). The DSLs most widely used by programmers are languages like HTML or SQL, but usually when we talk about DSLs in the context of synthesis, we mean ones that are even more narrowly focused. These DSLs tend to feature limited control flow and lack Turing completeness—instead, they pack all of the domain's complexity into a few carefully-selected operators, like the FlashFill DSL and its powerful regular expression matcher. This is necessitated by limitations in inductive synthesis technology; current approaches all have the flavor of enumerative search over the program space. Some exploit additional insights about the semantics of the language, or dispatch synthesis goals to SMT solvers like Z3 [27] (a more expressive kind of SAT solver), but the performance characteristics here are roughly the same: the more expressive and general-purpose the language, the wider and more intractable the program space. It is only by restricting the problem domain and applying specialized optimizations that researchers have been able to build synthesizers that scale to problems of practical interest [36].

Again returning to our spreadsheet example, suppose our synthesizer uses the following toy DSL. The annotation `@input` identifies the name of the input variable, and `@output` identifies the start token of the grammar (i.e. the expression that evaluates to the output value). The operator `split-nth` splits by whitespace and returns the nth element.

```
@input x : string
@output a : string ::= x | s | append(a, a) | split-nth(x, n)
s : string
n : int
```

In order to compute our desired function, the synthesizer might then generate the program: `append(append(split-nth(x, -2), " "), split-nth(x, 0))`. Without providing the second row as an example, it might have emitted the following program, which picks out middle initials instead of last names: `append(append(split-nth(x, 1), " "), split-nth(x, 0))`.

Drawing from lessons learned during the creation of FlashFill and FlashExtract, Polozov & Gulwani generalize the methodology for creating domain-specific inductive synthesizers into a framework called FlashMeta [65], later publicly released as PROSE [10]. Whereas previous synthesizers for particular domains would be implemented on an ad hoc basis, PROSE shares the underlying synthesis engine across domains, requiring the developer only to write a DSL specification—consisting of a syntax, a semantics, a ranking function for candidate programs, and *deductive inference rules*, also called *witness functions*

or *backpropagation rules*, for each operator. Conceptually, deduction rules are inverse semantics: given an operator's output, they use knowledge of the operator to return its possible inputs, or information about its inputs if the operator is not strictly invertible. These domain-specific witness functions are the key insight that propels the performance of PROSE and other deductive synthesis engines into the realm of practicality.

However, despite the massive improvement in expert productivity that comes from abstracting away the underlying synthesis engine [65], tools such as PROSE remain relatively inaccessible to those who are not well-versed in programming languages and synthesis [79]. We identify three key challenges when specifying a domain for deductive synthesis, elaborated in Section 3.1:

1. *Designing DSLs*. Finding primitives that strike the right balance between expressiveness and synthesizer performance is challenging even for experts [66], let alone novices, and requires an understanding of the behavior of the underlying synthesis engine.

2. *Writing deduction rules*. Not only are these functions unfamiliar to programmers accustomed to forward semantics, but manually specifying them is tedious and error-prone—they differ from inverse semantics in subtle ways.

3. *Debugging*. When bugs inevitably arise, the opaqueness of the synthesis engine can make it difficult to identify the root cause.

We propose *Programming "Programming by Example" by Example* (PbE^2), a new interaction model that aims to abstract these tensions away entirely. PbE^2 lifts the PbE philosophy of specification by example up a layer, applying it to the developer's interaction with the synthesis development framework, rather than just the user's interaction with the resulting synthesizer. The developer communicates their intended domain by providing a corpus of programs written in a general-purpose language, and the framework responds with an inductive synthesis engine that has been specialized to the corpus's inferred domain. With this shift, we have recast the problem of designing and implementing a DSL optimized for synthesis, which requires programming languages expertise, into the task of writing code, which a capable programmer with domain knowledge should be able to perform—just as PbE turns the problem of writing code into the task of providing examples. PbE^2 addresses the aforementioned pain points in the following ways, elaborated in Section 3.2:

1. *DSL design is automated*. Expert-written algorithms that are aware of synthesis performance characteristics carry out this task, removing the burden on the developer. In the same way that synthesizers specified through PROSE can benefit without added effort from advancements in underlying synthesis technology, domains specified as corpora with PbE^2 can benefit from improvements in DSL inference algorithms.

2. *Witness functions are drawn from a standard library.* The user does not need to manually write any witness functions; deduction rules for their programs are composed automatically from primitives in the standard library, written by synthesis experts. This approach is roughly analogous to automatic differentation, which has increased productivity in machine learning research by pushing derivatives and backpropagation, which previously needed to be programmed manually, into the language runtime.

3. *Developers have fewer opportunities to introduce bugs.* Because DSL design and witness function implementation are handled automatically, most bugs at the interface with the synthesis engine are no longer relevant. The remaining bugs are due to either implementation errors in the program corpus—which is a more familiar and self-contained debugging task—or a mismatch between the developer's domain intent and the output of the inference engine, which might be resolved interactively.

This refactoring of responsibilities admittedly puts a high burden on the synthesis expert, who now has to articulate their insight about DSL design into a DSL inference engine, as well as author a library of language operators with forward and backward semantics—all while writing a comprehensive test suite to validate this functionality. In order to demonstrate the feasibility of this approach, we present VERSE (VERified Synthesis Engine), an extensible Coq implementation of the PbE^2 interaction model. VERSE comprises a simple functional language and its library of operators, a DSL inference engine, and a deductive synthesis engine with a similar top-down deductive search algorithm to PROSE. The end-to-end proof of correctness is not yet mechanized; this is an ongoing project. Implementing PbE^2 as a Coq development poses two key benefits:

1. *Formal guarantees for developers.* A developer using VERSE receives a strong guarantee about its behavior: VERSE will output a DSL that includes every program in the corpus and a synthesizer that is both complete and sound with respect to the DSL's program space. That is to say, for a given set of example input/output pairs, the synthesizer will output exactly those programs in the DSL which satisfy the examples—no more, no less.

2. *Formal guarantees for experts.* We provide a formal interface that allows synthesis experts to write new language primitives in a modular fashion, in native Coq. This enables experts to validate their code in the following ways:

   (a) *Stronger type checking.* Thanks to the flexibility (or perhaps rigidity) of Coq's dependent type system, we can report catastrophic failures in witness functions at compile time, rather than discover them during synthesis.

(b) *Termination*. Because Coq programs are guaranteed to terminate, we avoid classes of bugs where witness functions diverge on certain inputs, or where the synthesis engine dispatches witness functions in an infinite loop.

(c) *Compositional correctness*. By proving a local correctness theorem about the correspondence between a primitive's forward and backward semantics, the author enables VERSE to prove automatically the correctness of the entire system.

In short, this work makes the following contributions:

1. We propose the PbE^2 interaction model, where developers specify a domain for synthesis using a corpus of programs rather than a language specification. This method precludes three key usability challenges for non-experts: designing DSLs, writing deduction rules, and debugging. It has the additional benefit that under-the-hood improvements can result in increased synthesis performance or expressiveness without developer intervention.

2. We provide a Coq implementation of PbE^2 as VERSE. Once the proof of correctness is complete, VERSE will enable developers to produce formally verified domain-specialized synthesis engines without any programming languages background, and surface a clean interface allowing experts to add new verified operators in a modular fashion.

The remainder of the thesis is organized as follows: Section 2 situates the PbE^2 interaction model and VERSE artifact in the context of related work. Section 3 motivates the design of PbE^2. Section 4 describes at a high level VERSE's architecture, as well as its formal guarantees and the compositional structure of its correctness proof. Section 5 establishes the object representations used across multiple passes of VERSE, including its reified types, expressions, grammars, and data structures. With these preliminaries addressed, Sections 6 and 7 take a deep dive into the DSL inference and inductive synthesis components of VERSE respectively. Section 8 outlines the assumptions and discrepancies of the Coq development with respect to VERSE's presentation in the previous sections. Section 9 concludes and discusses directions for future work.

2. Related Work

There are four main directions of inquiry that we consider immediately relevant to PbE^2 and VERSE. In Section 2.1, we survey the history and variety of interaction models for synthesis, both facing the user and the developer. In Section 2.2, we look at works on inferring context-free grammars from examples, towards use in VERSE's DSL inference engine. In Section 2.3, we explore the synthesis technologies that

enable PbE, and discuss whether recent innovations lessen the need for domain specialization. In Section 2.4, we compare the formal guarantees offered by VERSE to those of prior works in synthesis, and draw parallels to other works using proof assistants.

For more exposition on synthesis beyond the topics discussed here, see the excellent survey by Gulwani, Polozov, & Singh [36].

2.1 Interaction Models

Most of the variation in interaction models for synthesis comes from the method of specification that the end user, rather than the developer, employs. The first synthesizers historically, such as Waldinger & Lee's for Lisp [85] or Büchi & Landweber's for finite automata [25], sit at one end of the spectrum, where users must provide complete logical specifications of their desired program's functionality. Because logical specifications were often no shorter than the program they described (and more difficult to write, for someone accustomed to programming), the alternative of *Programming by Example* emerged, with Hardy [37] creating an inductive synthesizer that could generate Lisp programs from single input/output pairs, and Biermann [20] and Summers [78] from multiple example pairs. The related interaction model of *Programming by Demonstration*, where the user provides step-by-step instructions on how to reach the output from the input, followed soon after; this model is well-suited to interactive specification using GUIs, as in Smith's PYGMALION [70]. Over the years since these foundational works, the gaps between these two extremes have been gradually filled in—for example, through Solar-Lezama's SKETCH [71], which completes holes in partial programs supplied by the user, and type-directed synthesizers [58, 63], which combine supplied helper functions to construct programs inhabiting the type specified by the user.

Subsequent work from the angle of human–computer interaction has refined these patterns, in particular sharpening the basic PbE model into the more usable form [43] we see in practice today. The key insight from Wolfman et al. [89] was to make the interaction *mixed initiative*, meaning that both the user and the computer initiate interaction. More concretely, the synthesizer can ask the user to provide the output for a particular input in order to disambiguate their intent; Mayer et al. [50] evaluate how best to select and phrase these questions. Peleg et al. [61] enhance this model by allowing the user to give more granular feedback, like rejecting parts of candidate programs, rather than just providing additional examples. Le et al. [45] reintegrate these insights into a formal synthesis model. How to augment PbE^2 in a similar way—presenting inferred DSLs to developers in a digestible way, clarifying domain intent with computer-initiated questions, etc.—are interesting problems that we leave for future work.

In contrast to user-facing interfaces, developer-facing interfaces are more uniform. As we've already discussed, Polozov & Gulwani's FlashMeta [65]/PROSE [10] require the developer to provide a syntax, a semantics, and deduction rules—because the synthesis engine is deductive. Bodik & Torlak's Rosette [80]

is a competitor whose synthesis engine uses symbolic execution and SMT solvers rather than deduction, so its DSL specification comprises only a syntax and semantics. Wang et al.'s BLAZE uses abstraction refinement, so it asks for a syntax, a semantics, and suitable abstractions (explained in the next paragraph). The Syntax-Guided Synthesis paradigm [13] closely and intentionally mirrors the interface of SMT: while there is no domain-specialization ahead of time, the syntax and semantics of the DSL are specified in the synthesis query. These four interfaces improve expert productivity by disentangling the domain from the synthesis algorithm, but they hew pretty closely to the immediate demands of the synthesis engine.

Perhaps the closest work to PbE^2 is Wang et al.'s "Learning Abstractions for Program Synthesis" [86], which builds upon the BLAZE engine by inferring abstractions from example input/output pairs, of the kind that the end user would provide. At a high level, these abstractions are sound approximations to the original language with a smaller program space; for example, we might represent integers only as negative, zero, or positive, rather than with their actual values. We can use these coarse abstractions to prune away regions of the program space with less computational expense. Wang et al.'s implementation, ATLAS, learns abstractions that perform better than those manually specified by synthesis experts. A direct analogue of this work in the deductive context would be to learn witness functions automatically, not the DSL itself, but the idea is similar to PbE^2 in spirit. To the best of our knowledge, PbE^2 is the first work that proposes specifying domains for synthesis using program corpora.

2.2 Grammar Inference

Grammar inference (also known as grammar induction) is a well-studied area in learning theory [76]. We will focus on inference of context-free grammars (CFGs), as this format is most relevant to deductive synthesis, though historically these works were motivated by natural language. As a definitional note, when we say that an algorithm has "learned a CFG", we mean that the language (set of sentences) derived by the hypothesis CFG is the same as the ground truth context-free language. This is relevant when the same language can be represented by multiple CFGs; it suffices to learn any of them.

In a seminal work, Shamir [69] proves that it is impossible to learn context-free grammars from finite sets of *positive* examples (grammatical sentences); intuitively, for any language with an infinite number of sentences, a finite number of examples cannot disambiguate between the trivial grammar containing all the examples and the true grammar. Gold [32] shows us that our outlook is not quite as grim as it might have seemed: if we add *negative* examples (ungrammatical sentences), we can identify CFGs in the limit. That is to say, there is an algorithm that converges to the correct CFG in a finite number of examples.

Subsequent works follow with this trend, augmenting the learning model in order to circumvent the impossibility result. Knobe & Knobe [41] propose a simple method for inferring CFGs given access to a *teacher* who can answer membership queries. Their algorithm tries to infer the *most general* possible

grammar, in the sense that each rule we add to accommodate a positive example should accept as many other sentences as possible; it then prunes away overgeneralizations using negative examples. A natural question to ask is whether we can learn CFGs *efficiently* using this model, as prior results were just about decidability. Angluin's *minimally adequate teacher* (MAT) model gives this oracle even more power—now it checks whether the hypothesis grammar is equivalent to the ground truth language, and returns a counterexample if not—and this is enough to learn regular expressions in polynomial time [16], but sadly not context-free grammars [17]. Sakakibara [67] finds that CFGs are efficiently learnable when the MAT model is enriched with *structured* positive examples, which consist of the grammatical sentence and an unlabelled derivation tree. Oates et al. [57] investigate the learnability of CFGs from positive examples with a different kind of structure: *lexical semantics*, i.e. the linguistic meaning of each terminal node. Here, it is largely the type information that is helpful for inference, as the resulting grammar is constrained to be well-typed.

Langley & Stromsten [42] relax the learning objective, looking only to learn a CFG from positive examples that is approximately correct. Their strategy is to find a CFG that both satisfies the examples and has short description length. There are strong theoretical justifications for using Occam's razor as an inductive bias, dating back to the influential Blumer et al. [21]; concise hypotheses are more likely to generalize to unseen data, and the ability to explain the training set with a concise hypothesis implies PAC learnability. This approach is akin to rule-based compression algorithms like Neville-Manning & Witten's Sequitur algorithm [55], which achieve compression by inferring a concise CFG whose language contains only the given string. Sequitur was one of the works that originally inspired us to create PbE^2.

Črepinšek et al. [83] observe that most prior algorithms, analyzed theoretically or evaluated with small synthetic benchmarks, are ineffective on real programming languages. They manage to infer empirically the syntax of several small (but real) domain-specific languages using a genetic algorithm. Genetic algorithms and other similar nonconvex optimization techniques are a common way to learn CFGs in practice.

Our circumstances in PbE^2 are even more favorable than in most of the works above, making our inference task easier in the following ways:

1. Our positive examples have a great deal of structure, because they are embedded in a general-purpose language: they are well-typed abstract syntax trees with semantics.

2. For synthesis, the developer cares about the semantic program space rather than the syntactic program space. It is acceptable for the inferred DSL not to contain the exact programs the developer has in mind, as long as it can express semantically equivalent programs. It can be preferable to do this as an optimization, e.g. to reduce redundant encodings of the same functionality.

3. It is not especially important for us to reconstruct the "true" grammar perfectly.

   (a) Learning a domain that is more restrictive than intended is sometimes necessary, due to limitations in deductive synthesis technology: the best we can hope for is the projection of the true grammar into the space of synthesizable DSLs.

   (b) Learning a domain that is too general only degrades performance, preserving the functionality of the synthesizer.

As their algorithms are not directly applicable to our more relaxed learning model, we synthesize the key ideas of prior works into a simple, ad hoc inference algorithm (see Sections 4.1.2 and 6.3). Our treatment of learnability is informal, with our inference algorithm guaranteeing only that it preserves semantic membership of programs in the corpus (see Section 4.2).

2.3 Synthesis Engines

There are too many synthesis algorithms to do them all justice in one section. We will therefore explain the two most relevant to this work: bottom-up enumerative synthesis and top-down deductive synthesis. See Section 4.1 for their concrete applications in VERSE. Both of these techniques (as described here) work over a context free grammar where the start symbol represents a program with the output's type, not the type of a function from the input to the output; the input is represented as a bound variable with a particular name. That is to say, programs are of the form $x + x$ rather than $\lambda x.\ x + x$.

Bottom-up enumerative synthesis, in its naive form, brute forces the synthesis problem. Starting from the terminal nodes of the grammar (the "bottom"), we build up every possible program in the language, checking if it satisfies the specification. In the case of synthesis from examples, this means that for every provided example, the program maps the input to the output. A natural optimization is to cluster programs into equivalence classes based on their input/output behavior as we enumerate the space, e.g. by treating $x + x$ and $2x$ as the same subexpression, we prune the size of the program space. Alur et al. [14] and Udupa et al. [81] are examples of works that use bottom-up enumeration, albeit as a subroutine in more complex algorithms. We use bottom-up enumeration in the optional final passes of VERSE (see Sections 4.1, 5.4.3, 7.3, and 7.4), in order to transform a concise representation of synthesis output into a more readable list of programs.

Top-down deductive synthesis, in contrast, begins at the start symbol of the grammar (the "top") and works inward. For each alternative on the right hand side of the rule, from the specification for the entire expression we deduce necessary (and ideally sufficient) specifications for its subexpressions, and then recursively enumerate deduction over these. In synthesis from examples, this outermost specification is the

example pairs: for example, if the start rule of our grammar is `substring(x, N1, N2)` and we have the single input/output pair `("foo", "o")`, we deduce specifications for `N1` and `N2` such that `substring("foo", N1, N2) = "o"` and recurse. There are two main strategies for accomplishing deduction: inverse semantics and types. The former approach, popularized by FlashFill [35] and PROSE [65], uses handwritten per-operator inverse semantics to deduce possible input values from the specified output value, while the latter, underpinning MYTH [58] and SYNQUID [63], uses type information. Like with bottom-up enumeration, we can cluster subexpressions into equivalence classes based on their input/output behavior and use the resulting dynamic programming table to prune away some recursive calls. This data structure, called a *Version Space Algebra* (VSA), was originated by Mitchell [52]. We implement top-down deduction using VSAs as VERSE's core synthesis algorithm (see Sections 4.1, 5.4.1, and 7.1).

Common to both of these approaches—as well as the many others we omit—is that synthesis is feasible for programs up to a certain constant depth, usually between 15 and 30 abstract syntax nodes. This falls out as a consequence of exponential runtime: synthesis implicitly traverses the program space, which has size exponential in the length of the program, and pruning corresponds only to a reduction in the base of the exponent. A good DSL should therefore have concise representations for the functionalities we want, so that they fit inside the region that is within reach for current synthesis technologies. PbE^2 aims to make this domain specialization task, which is directly tied to synthesis algorithms, easier for non-experts. By embedding the inferred DSL within a general-purpose language, we are picking out the subset of the original program space that is most relevant to the task at hand.

Perhaps the most appealing alternative to making it *easier* to domain-specialize synthesis engines is making it *unnecessary* to do so. Although synthesis is undecidable in the general case, there is still room to push the boundaries of what we can achieve in practice. The first and rarer way to do this is with hard insights from programming languages theory. For example, van Tonder & Le Goues [82] and Polikarpova & Sergey [62] manage to synthesize imperative programs with pointers by casting the problem as proof search through separation logic. By doing so, they recover structural constraints that are missing from the syntax of imperative languages (e.g. preventing pointer mishaps) and eliminate meaningless regions of the program space. Advancements like these tend to expand the collection of language primitives that we can use in practical synthesis, but not improve the state of affairs for programs expressible in simpler languages. As we may now want to domain-specialize imperative languages for improved performance, this provides even greater impetus for models like PbE^2.

Second, the dominant trend in recent work on program synthesis has been the integration of machine learning: there is a rapidly growing body of research that aims to circumvent the need for domain specialization by augmenting traditional enumerative or deductive synthesis techniques with guidance from learned models, usually neural networks [18, 46, 64, 84, 90]. We can view this as an instantiation of the

general reinforcement learning strategy of neural-guided tree search. This approach is relatively orthogonal to domain specialization through PbE[2]: in some sense, neural guidance performs domain specialization at runtime, which can supplement domain specialization performed ahead of time through careful choice of DSLs. Furthermore, we could use machine learning within PbE[2] to infer a DSL from the provided corpus, which might be preferable to using neural networks at runtime because a) model execution is not free and b) DSLs and their performance characteristics under enumeration are more interpretable than a black-box guiding function. Either approach could be integrated within an end-to-end verified pipeline, as long as the learned model is used only to rank a set of valid options.

2.4 Formal Verification

In Section 2.4.1, we briefly explain what it means to formally verify a program in a proof assistant and draw parallels between our strategy with VERSE and other verified developments. In Section 2.4.2, we examine the interplay between verification and synthesis.

2.4.1 Verification in Proof Assistants

In order to gain increased confidence in the correctness of a program, or more generally the correctness of a mathematical theorem, one can write *machine-checkable* proofs. In contrast to hand-written proofs, which are written in English and usually require the human reader to fill in low-level details, machine-checkable proofs are represented computationally, such that a small, thoroughly-vetted program called a *kernel* can efficiently validate them. *Proof assistants*, also known as *interactive theorem provers*, are applications that help the user construct such proofs; the most prominent examples are Coq [26], Agda [49], and Isabelle/HOL [56]. These are an intermediate step towards the holy grail of *automated theorem provers*, which are essentially proof synthesizers.

There are two main approaches one can take when verifying a system in a proof assistant. The first is to write the system in a traditional language, and state the correctness theorem in terms of a formalization of that language's semantics. For example, this is the approach taken by CertiKOS [34], a concurrent OS kernel written in C and x86 assembly but verified in Coq. The second approach, which we adopt for VERSE, is to write the system in the native language provided by the proof assistant; in the case of Coq, this is a purely functional language called Gallina. The most influential example of this approach to us was Leroy's CompCert [47], an end-to-end verified optimizing compiler from a C-like language to several assembly language targets. VERSE is informed by many of Leroy's insights about architecting a verified compiler, especially the use of many single-purpose passes between representations with syntactically-encoded invariants. Even the organization of this paper, in particular Sections 4–7, is inspired by Leroy's journal paper describing the development and verification of CompCert [48].

2.4.2 *Verification and Synthesis*

Verification and synthesis are related at a fundamental, complexity-theoretic level: synthesis is the search version of verification's decision problem. The earliest synthesizers, like Waldinger & Lee [85] and Büchi & Landweber [25] mentioned above, were built on top of automated theorem provers. Because these systems use constructive logic, once we find a proof that there exists a program satisfying some specification, we can extract that program from the proof. Later works like Srivastava et al. [75], Frankle et al. [31], and Scherer [68] have continued to draw from the literature on verification and intuitionistic logic. These and many other synthesizers therefore generate programs that are "correct by construction", or "provably correct"—but the confidence we have in this soundness guarantee (that the program output by the synthesizer is correct) depends on the proof environment and implementation.

There is a spectrum of rigor that one can apply when deriving these correctness results. On one end are synthesizers whose algorithms are proven correct on paper and implemented in an ad hoc, unverified fashion. Though this is the "least rigorous" end of the spectrum, we already have relatively high confidence in the stated theorems; if there are errors, they are hopefully just implementation bugs. On the other end are synthesizers whose implementations are mechanically verified. This is the highest level of confidence that we can (currently) achieve in stated theorems, though it may still be the case that there are mistakes in the formal specification. And there are obvious downsides to mechanical verification—primarily, the immense implementation effort to verify anything of reasonable scale, which is all too familiar to us after working on this paper.

A natural compromise between these two extremes is to use an off-the-shelf, high-assurance program as a subroutine in a larger algorithm; this algorithm is then proven correct on paper, conditioned on the correctness of the subroutine. The most popular example of this is *counterexample-guided inductive synthesis* (CEGIS), originated by Solar-Lezama et al. for SKETCH [71, 72], building on Clarke et al.'s *counterexample-guided abstraction refinement* (CEGAR) for model checking. In the CEGIS paradigm, a synthesizer and verifier take turns: the synthesizer produces a candidate program, the verifier finds a counterexample to the desired specification, the synthesizer responds with a modified candidate, and so on. Synthesis ends when no counterexample exists. In practice, the synthesizer and/or verifier are instantiated as SMT solvers. Although standard CEGIS is not typically used for programming by example, as the verifier's job would be trivial, variants like Jha et al.'s *oracle-guided synthesis* [40] are still relevant. Here, the verifier disambiguates the specification in addition to checking for soundness; if there exist candidate programs that satisfy the examples so far but compute different functions, the verifier extends the specification with a new example on which they differ. In PbE, these new examples would be supplied by the user in a mixed-initiative interaction.

We might hope that we could adapt CEGIS et al. to be provably sound without trusting off-the-shelf

solvers, but this is usually not possible. While we can efficiently validate counterexamples provided by the verifier, we don't know how to efficiently validate the verifier when it says that no further counterexamples exist. We have good reason to think this problem is impossible in general: a solution would be proof that NP = co-NP. While traditional SAT solvers like MiniSat [29] can be implemented in as few as 600 lines of source code, modern SMT solvers like Z3 [27] consist of more than 450,000 [7], and although not all of these lines are correctness-critical, this does translate into a disconcertingly large number of correctness bugs [5]. In this sense, it is unclear how confident we should be in the correctness of these algorithms' implementations, especially when the harness connecting the solvers and translating the SMT input language is itself unverified. This is not to put Coq on a pedestal—it has experienced its fair share of soundness bugs [1]—but because proof assistants cleanly separate their trusted kernels from the rest of their implementation, mechanized proofs are generally considered to offer the strongest assurance of correctness.

We are aware of only one prior work on synthesis whose soundness is guaranteed by a proof assistant—but first, some context. *Interactive synthesizers* trade off automation for flexibility in much the same way as interactive theorem provers. Whereas in the models of Section 2.1, interaction is used to tease out the user's intent, here it is used to guide synthesis, in a similar role to the neural networks of Section 2.3. Itzhaky et al.'s Bellmania [39] is an interactive synthesizer that enables developers to write efficient and correct parallel implementations of dynamic programming algorithms; an SMT solver verifies the soundness of each interactive step. Delaware et al.'s Fiat [28] is a library supporting refinement of declarative specifications into functional programs in Coq; here, the Coq kernel validates each interactive step. Therefore, interactively synthesized programs are guaranteed to be as correct as the Coq kernel, though this soundness theorem cannot be meaningfully stated in Coq itself.

Much less common are formal guarantees about the completeness of synthesizers, in large part because program synthesis is undecidable in general. But when we limit the expressivity of the language and specifications, as in the works on deductive synthesis leading up to PROSE, we do expect that our synthesizer should generate every compliant program in our domain (perhaps modulo symmetry, etc.). The original presentation of FlashFill leaves completeness as a conjecture [35]; FlashExtract sketches a proof of completeness for its particular DSL [44]; and FlashMeta proves only soundness [65].

In this work, we sketch the architecture of VERSE's end-to-end correctness proof—this includes the soundness of VERSE's inference algorithm, as well as the soundness and completeness of its top-down deductive synthesis engine—which has been designed to be amenable to mechanical verification. In particular, we formally characterize the properties required of DSL operators to achieve correct synthesis. We are not aware of any prior works that mechanize either grammar induction or automatic (i.e non-interactive) synthesis in a proof assistant.

3. Design

In this section, we walk through the design process that led to several key decisions in PbE^2. In particular, we provide evidence that specifying domains *by example* meaningfully improves upon manual DSL specification, especially for developers with less expertise.

3.1 Challenges

In prior work [79], we investigated the challenges that novice users empirically face when attempting to use PROSE [10], the state of the art deductive synthesis framework. We performed a small exploratory user study on three college seniors, all of whom had software engineering experience but lacked formal exposure to program synthesis. After an introductory presentation, they were given access to PROSE documentation and tasked with completing an exercise adapted from PROSE's "DSL authoring" tutorial [6]. Essentially, they were provided with a small string-manipulation DSL and asked to add support to its `substring` operator for negative indices (which work modulo the length of the string)—see Figure 3.1.

Despite the structured nature of the assignment, the subjects found it difficult to conceptualize the solution to the problem. They noticed that the language's forward semantics for `substring` already accounted for negative indices, and decided that a change should be made to the witness function for `substring`, but none succeeded in making the correct change during their allotted hour. The solution accomplishes this with one additional line of code, which includes the negative equivalent of any specified positive index. Based on observation and discussion during the exercise, the subjects found this task difficult both because they were unfamiliar with this inverted, nondeterministic way of thinking, and due to practical difficulties interpreting the effects of any code they added. For more detail and experimental materials, see Tanzer & Wendland [79].

Based on these findings, the self-reported experience of others, and our own experience using PROSE, we identify three key challenges when specifying a domain for synthesis.

3.1.1 Designing DSLs

Synthesis experts underscore in presentations of their work the difficulty of designing performant DSLs [22, 66]. (By performant, we mean that synthesis over the DSL space is performant; for most synthesis use cases, performance of the synthesized program is a distant second priority behind correctness.) Though some of the intuition about what makes a good DSL can be carried over from human-facing DSLs like SQL, many of the hallmarks of a good DSL for synthesis are different and informed by the underlying synthesis algorithm being used.

```
// Toy DSL grammar
@input string x;
@start string program := Substring(x, pos, pos);
int pos := AbsPos(x, k);
int k;

// AbsPos forward semantics
public static int? AbsPos(string x, int k) {
    return k > 0 ? k - 1 : x.Length + k + 1;
}

// AbsPos deduction rule
[WitnessFunction(nameof(Semantics.AbsPos), 1)]
public DisjunctiveExamplesSpec WitnessK(GrammarRule rule,
                                        DisjunctiveExamplesSpec spec) {
    var kExamples = new Dictionary<State, IEnumerable<object>>();
    foreach (var example in spec.DisjunctiveExamples) {
        State inputState = example.Key;
        var x = inputState[rule.Body[0]] as string;

        var positions = new List<int>();
        foreach (int pos in example.Value) {
            positions.Add((int)pos + 1);
            positions.Add((int)pos - x.Length - 1);
        }
        if (positions.Count == 0) return null;
        kExamples[inputState] = positions.Cast<object>();
    }
    return DisjunctiveExamplesSpec.From(kExamples);
}
```

Figure 3.1: Excerpts from the PROSE tutorial [6] we used in our exploratory user study. This is the completed solution code; the prompt is identical except that it removes line 24.

The top snippet is the grammar specification, for a language that can compute a single substring of the input.

The middle is the forward semantics for `AbsPos`, an abstraction for indices into the substring that facilitates positive and negative constants as indices.

The bottom is the deduction rule for `AbsPos`. The developer must provide such a witness function for each argument to the operator. (Here only argument 1, k, is needed because v is fixed by the grammar.) Whereas the forward semantics is computed directly in terms of its arguments, the witness function uses several data structures that represent the program state abstractly.

For example, because engines like PROSE use a combination of deduction, enumeration, and dynamic programming, we generally want programs in the DSL to have an enforced canonical form, with few (if any) redundant encodings for the same program, as a kind of ad hoc symmetry reduction [22]. Where general-purpose languages tend to have flexible grammars with a small number of rules (e.g. expressions and statements) and arbitrary nesting, synthesis DSLs like FlashFill [35] have more rigid structure and are often bounded depth. The language operators in these DSLs should have relatively few parameters, each ideally within a finite universe, and be at an appropriate granularity for deduction (e.g. control flow like loops should be highly structured and packed inside of the operator). If the user is designing a DSL for a different synthesis framework—like Rosette [80], which performs synthesis using symbolic execution and an off-the-shelf SMT solver (currently Z3 [27])—design decisions for the DSL may also be informed by the theories supported by the solver or the heuristics used to pick between them, as aligning with these grants a substantial boost to performance [88].

DSL design is a skill that can be (and has been) taught to those with sufficient background [22, 23, 24], but it's not necessarily one that software engineers currently possess.

3.1.2 Writing deduction rules

We witnessed in our user study that software engineering background does not imply comfort with inverse semantics, and that beyond just the conceptual difficulty of inverse semantics, there is cognitive overhead from working with an abstract representation of program state (as in Figure 3.1). While this way of thinking is natural to programming language researchers, it may be unfamiliar to those who use programming merely as a tool rather than an object of study.

Furthermore, PROSE's witness functions are not strictly inverse semantics: their interface often coincides with inverse semantics but is really determined by the function they serve in the underlying synthesis engine. For example, witness functions must be defined such that the synthesis engine's traversal of the program space will exhibit well-founded recursion. This requirement was not made apparent to us from the definition of witness functions in the FlashMeta paper [65] or PROSE documentation [10], but rather was clarified through personal communication with a developer of PROSE [9] after our toy synthesizer diverged.

We were attempting to implement a small DSL with only a string `append` operator, which could be nested an arbitrary number of times. The forward semantics were trivial, using C#'s native string append operator. Our witness functions for `append` strictly inverted these semantics, e.g. for the output "foo", we returned the input pairs ("", "foo"), ("f", "oo"), ("fo", "o"), and ("foo", ""). But because of the granularity at which PROSE memoizes deduction, returning ("foo", "") or ("", "foo") will cause deduction on "foo" recursively, ad infinitum. In order to resolve this, we either omit input pairs with "" or bound the depth

of recursion in our grammar.

In many cases this is a sensible restriction, as the additional bits we miss out on (here appending empty strings) are no-ops, though one can imagine situations where these no-ops could become meaningful, like if an expression evaluates to an empty string on some inputs but not others (e.g. append a middle initial if one exists). But in order to anticipate this, the programmer must understand not only top-down deductive synthesis in general, but also PROSE's specific implementation of it. (For instance, VERSE happens to handle our `append` example correctly, because it memoizes at the granularity of witness function invocations.)

3.1.3 Debugging

Both of these challenges are compounded by the absence of feedback about developer mistakes. PROSE lacks static checks for many aspects of its interface: containers are dynamically cast to and from `objects` at the boundaries of each witness function, variables are accessed through dynamic lookups, and coverage of the grammar with witness functions is only reflected at runtime. If you're lucky, a mistake in one of these areas will result in a runtime exception, but much of the time it results in an empty—or even worse, incomplete—list of programs synthesized, perhaps only for certain input/output examples.

In our experience, the most effective way to debug these problems is to instrument witness functions with copious printouts in order to trace the path that the synthesis engine takes through the deductive space. When a witness function fails to be invoked, however, this method provides no information. For example, if you take an *ExampleSpec* as an argument rather than a *DisjunctiveExamplesSpec*, there are situations where only the outermost layer of an operator that is supposed to be nestable synthesizes correctly, e.g. `append(v, v)` appears but not `append(v, append(v, v))`. This error—the absence of a suitable witness function—is only visible in the incompleteness of synthesis output.

These stumbling blocks likely present no issue to a PROSE expert, but they can be frustrating for beginners. Even if these particular inconveniences can be solved with static analyses, the underlying problem—debugging a callback function without understanding when or why it's invoked—still remains.

3.2 Approach

The *Programming "Programming by Example" by Example* model responds to these challenging tasks essentially by avoiding them altogether. Because the developer needs only provide a corpus of programs, they don't have to understand the nuances of designing DSLs and writing deduction rules, or debug the errors that would have resulted if they had tried to do so with an incomplete understanding.

In Tanzer & Wendland [79], we performed a small confirmatory user study on a simulated implementation of PbE^2 to sanity check our approach. The participants were four college seniors, with similar

```
@input string x
@output string

append(x, substring(x, 0, 2))
append(append(x, x), x)
```

```
@input x : string
@output a : string ::= append(b, s)
b : string ::= x | a
s : string ::= x | substring(x, n, n)
n : int
```

Figure 3.2: An example solution for our confirmatory PbE^2 user study.

The left is what the developer provides: a corpus consisting of input/output type annotations and two programs, which use append and `substring` from the standard library. Longer corpora should in principle give the inference algorithm a better sense of the domain.

The right is a DSL that might be inferred automatically from the corpus. (Note that both programs on the left can be derived by this grammar.) This DSL could be slotted directly into the synthesis engine, or fine-tuned first if so desired. The way that this corpus generalizes into a DSL depends on the underlying inference algorithm, and in a real product should use a mixed initiative interface, e.g. to decide whether 0 and 2 are special constants, representative of `nats`, or representative of `ints`.

demographics to our exploratory study. After an introductory presentation, they were given access to VERSE documentation and tasked with making a synthesizer for the domain of string transformations using `substring` and `append`, with only absolute position indices. In order to make the task nontrivial, we communicated this domain to them using input/output examples and included tens of unneeded operators in our documentation. We tried to make the study's task similar in scope to that for PROSE, but they are somewhat incomparable because of differing abstraction levels. See Figure 3.2 for an example solution.

3 out of the 4 subjects successfully completed this task in the allotted hour; the subject who failed to do so also demonstrated conceptual misunderstanding of program synthesis after the introductory presentation. These results are encouraging for PbE^2, but it remains to be seen whether this scales to more complex tasks. It is a challenging research question to design meaningful and cost-effective user studies for tools that take multiple sessions to learn, or for tasks of moderate length. We expect that PROSE should take time to learn, so its study is more useful for identifying barriers to entry, rather than as a direct comparison. Due to the limited scope of our experiment (lack of ablation), we also cannot say which of PbE^2's design elements are responsible for its improved usability. We will now discuss why we prefer specification by corpus to some plausible intermediate alternatives.

A reasonable first step towards improved usability would be to augment PROSE with a robust standard library of deduction rules. PROSE's collection of "standard concepts" [8] is a start at this, albeit currently limited to primitives for functional control flow and manipulating algebraic data types, like mapping a function over a list and constructing a pair. However, even though this approach addresses the challenges of writing deduction rules and DSL design (at least in the sense of defining language primitives), constructing a grammar that composes these witness functions correctly is a nontrivial task. Witness functions are often limited in ways that only make sense when you understand their implementation,

which makes them difficult to use across an abstraction barrier. For example, the `Substring` and `AbsPos` operators in Figure 3.1 must be bound to a particular string. This is compounded by the issues with debugging discussed in Section 3.1.3 above, as malformed grammars will be reflected only in incomplete synthesis output at runtime. By using a corpus as a form of specification, we delegate this knowledge of witness function idiosyncrasies to the inference engine, which complies by generalizing the example programs to greater or lesser extents. From the perspective of the developer, the resulting synthesizer is as general as the underlying synthesis technology permits.

We might simplify the task of writing a grammar instead by asking the developer to select the set of operators they want present in their DSL: from this set—a "bag of operators" by analogy to "bag of words" models—the inference algorithm would construct a grammar where operators combine with each other as flexibly as types allow. This approach has the obvious disadvantage that the program space is unnecessarily wide, because the domain specification lacks the structure present in a corpus. But the specification task of selecting operators initially seems easier than providing a corpus of programs. We are not convinced that this is actually the case. There is a distinction in epistemology between *knowledge-how* and *knowledge-that*, or in cognitive psychology between *procedural knowledge* and *declarative knowledge* [30]: the former refers to intuitive skills while the latter refers to conscious knowledge. Applied to this context, writing programs in a domain and abstractly reasoning about what operators would be present in those programs do not necessarily exercise the same faculties. We somewhat embarrassingly experienced this ourselves while testing the `append`-only DSL mentioned in Section 3.1.2 above. We were initially confounded when our synthesizer failed to generate a program mapping "foo" to "foo foo", and only realized that this function is out-of-domain when we attempted to write a program computing it ourselves. We therefore believe that writing a corpus of programs more closely aligns with the knowledge that domain experts already have, rather than declaring a set of operators—this could be experimentally verified in future work. We include this "bag of operators" model in our evaluation of `VERSE` in Section **??** as another point in the usability/performance tradeoff space.

Beyond the weakness of these particular alternatives, there are several reasons to support $\texttt{PbE}^2$ in its own right. First, developers may already collect corpora in order to understand consumer needs and validate their products; for example, Gulwani's original evaluation of FlashFill [35] uses a benchmark suite containing representative examples collected by the Excel product team. One would adapt such a corpus for $\texttt{PbE}^2$ with minimal effort by splitting it into training and test sets, as is common in machine learning. Second, decoupling domains from their DSL implementations can lead to performance gains. Padhi et al. [59] improve synthesis performance by interleaving grammars of increasing expressivity; this is easier when the domain specification is not tied to a particular grammar. And third, $\texttt{PbE}^2$ and manual DSL specification aren't mutually exclusive: in the worst case, a synthesis expert can use a corpus to

bootstrap manual DSL creation, by automatically inferring a DSL and then fine-tuning the language manually.

3.3 Parallels

We now draw what we hope is an instructive parallel between the contributions of PbE^2 and of automatic differentiation in machine learning.

Just a few short years ago, machine learning practitioners manually computed and implemented gradient formulas for each of their models in order to train them [19]. This burden was unnecessary; today, libraries like PyTorch [60] and TensorFlow [12] automatically compute gradients within the language runtime. As a program executes, a computation graph is assembled by composition of language primitives; when the gradient is requested, derivatives are propagated backwards along the constructed computation graph by the chain rule. This has proven a massive productivity and quality-of-life improvement for machine learning researchers, who can focus their time on more experimentation rather than rote derivations, and decreased the barrier to entry for beginners. While autodiff is limited in certain fundamental ways, like that it cannot extract useful information from nondifferentiable primitives like control flow, developers are happy to accept these limitations. Those who wish to circumvent these shortcomings implement separate algorithms like REINFORCE [87], often used in conjunction with autodiff.

At its core, PbE^2 also aims to shift unnecessary or redundant labor into the language toolchain. Rather than allow developers to define their DSLs or synthesizers in full generality, we restrict their options to a language consisting of a set of primitives and simple rules for combining them. These primitives have special-purpose backpropagation rules in order to achieve better synthesis, much like functions in machine learning libraries define custom, more efficient (with respect to performance or numerical precision) backpropagation rules, even when those gradients could have been computed by composition of other primitives. We hope that this compromise is sufficient to satisfy the needs of most developers, especially when taking into account those who would not have otherwise been able to create synthesizers, and that those users who *do* need to escape the language boundaries still save time by writing the rest of their synthesizer by example.

4. Architecture

In this section, we describe at a high level the passes in the VERSE pipeline and sketch out how our correctness proof is composed along them. These descriptions will be refined in Sections 5, 6, and 7.

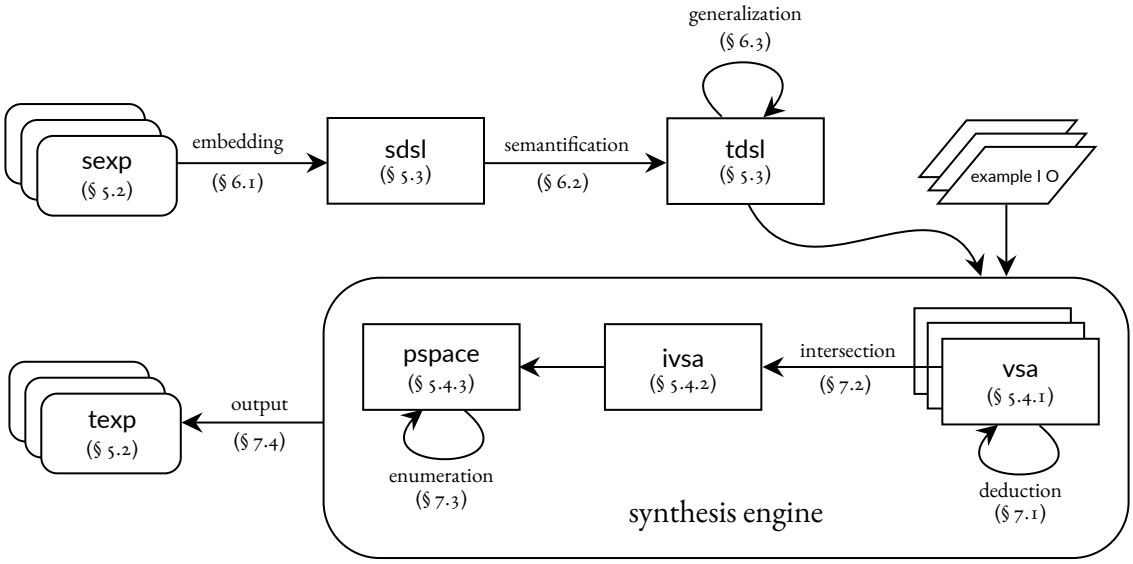

Figure 4.1: The passes and representations of VERSE, with inference in the top row and synthesis in the bottom.

4.1 System Architecture

We can group the passes of VERSE, depicted in Figure 4.1, into two categories: inference and synthesis. The inference passes are executed when the developer provides a corpus of programs, resulting in a DSL specification; the synthesis engine is partially applied to the DSL in order to domain-specialize it. The passes within the synthesis engine are executed later, when the end user provides input/output examples.

4.1.1 General-Purpose Language

The design of VERSE's general-purpose language (GPL) is mostly determined by the limitations of current deductive synthesis technology. The language itself is incredibly simple and has essentially no computational content: it is simply typed and monomorphic, supporting lambda abstraction but not function application. Its key feature is the ability to invoke named functions written natively in Coq, which are the units on which we perform deduction—we call these functions, operators, and primitives indiscriminately. These functions defined in Coq can perform arbitrary computations as long as they match the GPL's type system at their boundaries. This allows us to implement functionality ranging from string `append`, to `ifthenelse`, to `apply`—so that we can put the GPL's lambda abstractions to work. See the abstract syntax tree in Figure 4.2 for an example of how Coq functions are embedded in GPL expressions.

```
(F [TStr; TStr] "eq" [B 0; C TStr "foo"])
```

Figure 4.2: This program checks whether the input string and the constant string "foo" are equal. `"eq"` names a Coq **F**unction, which here takes two parameters of type `TStr`. These parameters are the **B**ound ID 0, which represents the input, and the **C**onstant string `"foo"`. This representation is intended to be used beneath a more pleasant user-facing language, which would reflect standard library operators like `"ifthenelse"` or `"letin"` as language primitives.

4.1.2 Inference

We will now describe the inference passes—embedding, semantification, and generalization—which collectively transform a corpus of programs in the GPL to a DSL specification.

Embedding

The embedding pass generates a context-free grammar that is the disjunction of the programs in the corpus. Our CFG representation restricts the size of each constructor to one abstract syntax node, so this pass also unflattens the expressions in a sense. For example, the left is the corpus of programs, and the right is their disjunction as a CFG:

```
@input string x
@output string

append(x, substring(x, 0, 2))
append(append(x, x), x)
```

```
@input x : string
@output a : string ::= append(b, c) | append(g, j)
b : string ::= x
c : string ::= substring(d, e, f)
d : string ::= x
e : int ::= 0
f : int ::= 2
g : string ::= append(h, i)
h : string ::= x
i : string ::= x
j : string ::= x
```

Semantification

The semantification pass looks up Coq functions in the standard library by name and embeds their semantics into the DSL. In a sense, this is the step that transitions from a general-purpose language to a domain-specific language, as the standard library is no longer needed. This pass could easily be collapsed with embedding, but distinguishing them simplifies the correctness proof.

Generalization

The generalization pass is where the actual inference happens. The result of the pass is a grammar with the following two properties:

1. There is one rule for each argument of each operator. For example, wherever append appears in the grammar, its first argument will be the same rule identifier.

2. Where possible, each rule has:

   (a) *No identical constructors*. Redundant constructors, including those made redundant by previous generalizations, will be removed.

   (b) *At most one constant constructor*. For example, if append("foo", x) and append("bar", x) both appear in the grammar, these will be generalized to append(s, x), where s is an unconstrained string.

The "where possible" caveat comes from operators where certain arguments must be finitely enumerable, which we will discuss shortly in Section 4.1.3. Recall substring, where the input string must be fixed during deduction; the rule identifier for this argument must therefore produce a finite number of strings, so that we can map deduction over them. In the semantification pass above, we translate operators in different contexts (unconstrained vs. finitely enumerable) with different aliases to make it easier for us to respect this rule.

To continue our example from above, generalization transforms the DSL on the left into the version on the right:

```
@input x : string
@output a : string ::= append(b, c) | append(g, j)
b : string ::= x
c : string ::= substring(d, e, f)
d : string ::= x
e : int ::= 0
f : int ::= 2
g : string ::= append(h, i)
h : string ::= x
i : string ::= x
j : string ::= x
```

```
@input x : string
@output a : string ::= append(b, c)
b : string ::= x | append(b, c)
c : string ::= substring(d, e, f) | x
d : string ::= x
e : int ::= 0
f : int ::= 2
```

Observe that the generalized grammar preserves membership of the two original programs from the cor-

pus, as no step in the algorithm may remove a sentence from the language, while introducing new programs, like arbitrarily left-nested appends.

4.1.3 Synthesis

We will now describe the synthesis passes—deduction, intersection, enumeration, and output—which collectively transform a set of input/output pairs to a list of all the programs in the DSL which satisfy those pairs. But first, a brief digression about the interface we require for deduction rules. We'll go through a series of representative operators to motivate these decisions. Recall that in top-down deductive synthesis, we want to deduce from the output what the inputs could possibly be.

1. For append, basic inverse semantics are enough: given an output string, there are a finite number of fully-determined input argument pairs.

2. For substring, this is no longer the case: the string argument must be fixed in order for deduction to succeed, since there are an infinite number of superstrings for any substring. We can resolve this with a sort of partial inverse semantics, where some prespecified inputs are used for deduction in addition to the output value. As long as this set of prespecified inputs is finite, we can generate them by forward enumeration and map deduction over them. We can ensure that the set of inputs is finite during construction of the DSL, as mentioned in Section 4.1.2.

3. For if then else, the output only gives us information about one of the branches taken, and by symmetry it could be either one. We want to be able to say that given the output v, the two possibilities are "if `true` then v else `unconstrained`" and "if `false` then `unconstrained` else v". So, we will make each deduced input argument an option type, where `None` represents an unconstrained value.

4. For apply or map, which work over anonymous functions defined in the DSL itself (and require the non-function argument to be pre-specified), the problem of inverting the semantics is precisely program synthesis from examples. If we required the program space to be finite, like PROSE does, the witness function could recursively call the synthesizer without special accommodation, since it is guaranteed to terminate. However, if we want to be able to stop the synthesizer early for performance reasons (e.g. only synthesize programs up to depth 6), having deduction rules that run an entire synthesis subroutine makes this difficult. If we add a special case for deduced input functions, which specifies the synthesis problem rather than solve it, we can interleave the recursive synthesis routine with the standard one. This has the added benefit that infinite program spaces are achievable, since we can choose to synthesize longer and longer programs.

Therefore, the behavior of an operator's deduction rule should be roughly the following: unfixed input arguments are deduced (either explicitly or as an unconstrained value) from an output if and only if the operator applied to those inputs evaluates to that output under the forward semantics. See Section 5.3.3 for a formal treatment of this.

Deduction

The *deduction* pass implements standard top-down deductive synthesis, as described in Section 2.3. Given a DSL and a collection of input/output examples, for each input/output example we construct a version space algebra (VSA) that concisely represents the program space satisfying the example. We do this essentially by performing breadth-first search on the space of programs (clustered by output value), starting with the example's output value and working backwards by deduction. In order to support infinite program spaces and early stopping, deduction is parameterized by the number of BFS iterations that should be performed; the complete VSA is the least fixed point of this, and represents the set of all programs satisfying a particular example pair.

Intersection

Conceptually, the *intersection* pass intersects the program sets satisfying each example, resulting in a single program set that satisfies every example. Intersection on VSAs, originated by Wolfman et al. [89] and used by PROSE [65], has a straightforward implementation: we traverse two VSAs simultaneously, and keep only the edges that are present in both. The resulting VSA can be manipulated in its current form (e.g. one could efficiently extract the top k programs according to some ranking function), or converted into an explicit list of programs by the remaining passes.

Enumeration

The *enumeration* pass implements bottom-up enumeration, as described in Section 2.3, in order to generate explicitly the set of programs represented implicitly by a VSA. We start from the terminal nodes of the intersected VSA and iteratively build up the set of programs associated with each nonterminal node; for unconstrained subexpressions (from operators like `if then else`), we enumerate the original DSL space instead. Again, this pass is parameterized by the number of iterations and the complete program set is the least fixed point.

Output

The output step is trivial: we return the set in the program space corresponding to the start rule.

4.2 Proof Architecture

In this section, we describe semi-formally the theorem characterizing VERSE's behavior. We will use the following canonical variables and notations, as well as source and **target** styles for elements when relevant.

$\mathbf{e}$	an expression
$\mathbf{L}$	a language/grammar (set of expressions)
$\mathbf{x}$	an input/output pair (example)
$\mathbf{X}$	a set of input/output examples
$\mathbf{e} \models \mathbf{x}$	the expression $\mathbf{e}$ satisfies the input/output example $\mathbf{x}$
$\mathbf{e} \models \mathbf{X}$	the expression $\mathbf{e}$ satisfies all examples in $\mathbf{X}$
$\mathbf{S_L}$	a synthesizer domain-specialized to $\mathbf{L}$

First, we want to define a kind of *sound generalization* for inference; the inferred DSL should "contain" our corpus in some sense. The most straightforward kind of generalization is syntactic, where every program in the source language must also be syntactically present in the target language:

Definition 1 *syntactic membership preservation* ($L_S \subseteq \boldsymbol{L_T}$)

$\forall\, e.\ e \in L_S \Rightarrow e \in \boldsymbol{L_T}$

This notion is too strong, as it doesn't allow for different representations for expressions, or optimizations that prune the program space without meaningfully affecting synthesis results (e.g. symmetry reductions). Therefore, we introduce a standard notion of semantic equivalence across source and target languages: expressions are equivalent if they have the same input/output behavior. (The semantics of expressions are defined using a large-step semantics.)

Definition 2 *semantic equivalence* ($e_S \approx \boldsymbol{e_T}$)

$\forall\, x.\ e_S \models x \iff \boldsymbol{e_T} \models x$

Now we use this to define the correct property, which requires only that the inferred DSL contain programs equivalent in behavior to the corpus.

Definition 3 *semantic membership preservation* ($L_S \sqsubseteq \boldsymbol{L_T}$)

$\forall\, e_S.\ e_S \in L_S \Rightarrow \exists\, \boldsymbol{e_T}.\ e_S \approx \boldsymbol{e_T} \wedge \boldsymbol{e_T} \in \boldsymbol{L_T}$

Note that syntactic membership preservation implies semantic membership preservation, and that semantic membership preservation is transitive: we can use these properties to compose the proof of correctness along VERSE's inference passes.

Next, we need to define correctness for the synthesizer. Here the guarantees are more conventional: soundness and completeness. Soundness means that every program that the synthesizer outputs satisfies the given examples (and is in the DSL):

Definition 4 *synthesizer soundness* ($S_L\downarrow$)

$\forall\, X, e.\ \ e \in S_L(X) \Rightarrow e \in L \wedge e \vDash X$

Completeness means that the synthesizer outputs every program in the DSL satisfying the given examples:

Definition 5 *synthesizer completeness* ($S_L\uparrow$)

$\forall\, X, e.\ \ e \in L \wedge e \vDash X \Rightarrow e \in S_L(X)$

This definition ignores the possibility of infinite program spaces for brevity—we will handle this detail in our more formal presentation below. Correctness is the combination of these two properties:

Definition 6 *synthesizer correctness* ($S_L\updownarrow$)

$\forall\, X, e.\ \ e \in S_L(X) \iff e \in L \wedge e \vDash X$

Our end-to-end correctness theorem for VERSE composes these two correctness criteria: the inference engine will output both a DSL that semantically preserves the membership of the programs in the corpus and a correct synthesizer specialized to this DSL:

Theorem 1 *VERSE correctness*

$\forall\, L_S\ \exists\, \boldsymbol{L_T}, \boldsymbol{S_{L_T}}.\ \ \mathit{VERSE}(L_S) = (\boldsymbol{L_T}, \boldsymbol{S_{L_T}}) \wedge L_S \sqsubseteq \boldsymbol{L_T} \wedge \boldsymbol{S_{L_T}}\updownarrow$

5. Infrastructure

In this section, we establish the representations used across multiple stages of VERSE in preparation for more in-depth discussion of its functionality and correctness later.

First, we have to get some notational details out of the way. In order to accommodate indexed type definitions and constructors with dependently typed arguments, we will use a slightly nonstandard variant of BNF that more closely mirrors the syntax of Coq; see Figure 5.1 for an illustration of this.

We will distinguish several identifier types in order to make immediate what they index into:

```
A (B : C) {D : E}  ::=  F  |  G
                        |  H (a₁ a₂ : A B)
                        |  I (b : B) (c : T b)
```

Figure 5.1: **A** is a data type parameterized explicitly by an object **B**, which itself has type **C**, and implicitly by an object **D** of type **E**. **F** and **G** are constructors that require no arguments. **H** is a constructor that requires arguments $\mathbf{a}_1$ and $\mathbf{a}_2$, both of type **A B** (implicitly, this is type **A B D**). **I** is a constructor that takes an argument **b**, of type **B**, and an object **c** with type **T b**, where **T** is a function from **B**s to types.

id	the name of an operator in the source general-purpose language
pid	the name of an operator in target domain-specific language
bid	a bound variable (including the input variable, which is bound to a canonical identifier)
rid	a rule in the CFG description of a DSL
did	an index into the structure storing recursive DSLs (for bodies of lambdas)
vid	a rule corresponding to a particular DSL rule and output value in the VSA data structure
sid	an index into the structure storing internal state for recursive synthesis of functions

We canonically refer to an identifier of type **xid** as $\mathbf{i_x}$. We will also make use of the following notations:

$\mathbf{T}^*$	the type of (possibly empty) lists of objects with type **T**
$\mathbf{T}^+$	the type of nonempty lists
[]	an empty list
[x]	a list with one element, **x**
x :: xs	a list with the head **x** and tail **xs**
$\mathbf{A} \rightharpoonup \mathbf{B}$	a finite map from **A** to **B** (not a partial function)
$\mathbf{A}^?$	an **option A**
$\lfloor \mathbf{a} \rfloor$	Some **a**
$\varnothing$	None

We will now describe the representations and properties of VERSE's reified types (§ 5.1), expressions (§ 5.2), grammars (§ 5.3), and data structures (§ 5.4). These definitions will be used in subsequent sections to formally characterize VERSE's behavior.

5.1 Types

The types of VERSE's general-purpose language are used at all levels of the pipeline: in inference, these types are present as annotations and used to state metatheory, while for synthesis, they are used as reified representations for native Coq types, which are difficult to work with directly because they cannot be pattern matched. (By *reified types*, we mean that types have concrete representations that are used at runtime, rather than abstract representations used only during compilation.) We implement a minimal type system for reasons of scope; making general reified types for Coq is a research topic in its own right [33, 73], as current metaprogramming infrastructure is limited.

The type of values, **vtype**, is defined as follows. We support booleans, natural numbers, integers, and strings, as well as product types (pairs) and lists.

```
vtype ::= TBool | TNat | TInt | TStr
        | TPair (V1 V2 : vtype)
        | TList (V : vtype)
```

Types in general are either a value type, or a function from a value type (the binder type) to a value type (the return type).

$$\begin{aligned} \mathsf{type} \ ::= \ & \mathsf{TVal}\ (\mathsf{V} : \mathsf{vtype}) \\ \mid \ & \mathsf{TFun}\ (\mathsf{V}_1\ \mathsf{V}_2 : \mathsf{vtype}) \end{aligned}$$

Note that this differs from e.g. simply-typed lambda calculus, where functions are values. This decision was made both due to limitations in deductive synthesis and out of a desire for simplicity; for example, it is more complicated to characterize inverse semantics of operators when functions may be returned, since syntactic equality will need to be replaced with alpha equivalence.

The above function type is only for lambda abstractions defined in the GPL; the interface for embedded Coq functions is more complex, and does allow functions as arguments. The type of these arguments is represented as nonempty list of types, **ftype**.

$$\mathsf{ftype} \ = \ \mathsf{type}^+$$

In order to integrate Coq programs into the GPL, we have to define a series of translations from our reified type system to native Coq (i.e. mathematical) types; we will refer to reified types as type and mathematical types as Type. We write reified types as T, their translated mathematical types as $\mathcal{T}$, and objects of the mathematical type as t. The first of these translations is **vraise** : vtype $\to$ Type, which raises value types into mathematical types.

$$\begin{aligned} \mathsf{vraise}\ \mathsf{TBool} &= \mathsf{bool} \\ \mathsf{vraise}\ \mathsf{TNat} &= \mathbb{N} \\ \mathsf{vraise}\ \mathsf{TInt} &= \mathbb{Z} \\ \mathsf{vraise}\ \mathsf{TStr} &= \mathsf{string} \\ \mathsf{vraise}\ (\mathsf{TPair}\ \mathsf{V}_1\ \mathsf{V}_2) &= (\mathsf{vraise}\ \mathsf{V}_1) \times (\mathsf{vraise}\ \mathsf{V}_2) \\ \mathsf{vraise}\ (\mathsf{TList}\ \mathsf{V}) &= (\mathsf{vraise}\ \mathsf{V})^* \end{aligned}$$

In the process of embedding the GPL's type system into native Coq, we also specified a decidable fragment of the mathematical type system. We can define a decision procedure with the type $\forall$ **V**. **vraise V** $\to$ **vraise V** $\to$ **bool**, and so on for the rest of the translations.

With **vraise** in hand, we can define the type of an example specification: a nonempty list of inputs and outputs, as objects with the raised mathematical types:

$$\mathsf{examples}\ \mathsf{I}\ \mathsf{O} \ = \ (\mathsf{vraise}\ \mathsf{I} \times \mathsf{vraise}\ \mathsf{O})^+$$

Next we need to define translations for type. The case for TVal follows from above, but the case for function types is more subtle—we actually need different translations for the forward and backward semantics. In the forward direction, we can translate to normal language expressions as expected, in **traise_fw** : Type $\to$ type $\to$ Type. (We parameterize by a Type $\mathcal{E}$ of expressions.)

$$\begin{aligned}\mathsf{traise\_fw}\ \mathcal{E}\ (\mathsf{TVal\ V}) &= \mathsf{vraise\ V}\\ \mathsf{traise\_fw}\ \mathcal{E}\ (\mathsf{TFun\ \_\ \_}) &= \mathsf{bid} \times \mathsf{vtype} \times \mathcal{E}\end{aligned}$$

But in the backward direction, the translation describes the outputs of deduction rules. As we discussed in § 4.1.3, there are several reasons to avoid performing recursive synthesis inside the deduction rule itself, like making it unwieldy to stop early or synthesize infinite program spaces, and inefficiency due to the lack of a concise data structure like VSAs. So, in **traise_bw** : type → Type, we translate function types into a specification of the synthesis problem: a list of examples.

$$\begin{aligned}\mathsf{traise\_bw}\ (\mathsf{TVal\ V}) &= \mathsf{vraise\ V}\\ \mathsf{traise\_bw}\ (\mathsf{TFun\ V_1\ V_2}) &= \mathsf{examples\ V_1\ V_2}\end{aligned}$$

We must also translate the type of argument tuples, which is more straightforward. We allow operators to fail by returning ∅, so we also make the arguments to operators option types (e.g. so that an error in the untaken branch of an if then else is not catastrophic). We parameterize **fraise** : (type → Type) → ftype → Type by a function to raise types to share structure across the forward and backward definitions.

$$\begin{aligned}\mathsf{fraise\ traise\ [T]} &= (\mathsf{traise\ T})^?\\ \mathsf{fraise\ traise\ (T :: Ts)} &= (\mathsf{traise\ T})^? \times (\mathsf{fraise\ traise\ Ts})\end{aligned}$$

Therefore, **fraise_fw** = **fraise** traise_fw and **fraise_bw** = **fraise** traise_bw.

More generally, we define a function, **fraiseT** : Type → ftype → Type, that allows us to create the type of tuples with the same structure as a particular ftype.

$$\begin{aligned}\mathsf{fraiseT}\ \mathcal{T}\ [\mathsf{T}] &= \mathcal{T}\\ \mathsf{fraiseT}\ \mathcal{T}\ \mathsf{traise\ (T :: Ts)} &= \mathcal{T} \times (\mathsf{fraise}\ \mathcal{T}\ \mathsf{Ts})\end{aligned}$$

5.2 Expressions

Expressions in VERSE's GPL are simply typed and monomorphic. Functionality in the language itself is extremely limited, with all the complexity being derived from embedded Coq operators, which respect the GPL's type system at the boundaries but are free to do as they see fit inside their bodies. Recall that we phrase programs as $x + x$, rather than $\lambda x.\ x + x$.

5.2.1 Syntax

We use a single data type for the abstract syntax trees of expressions across both the source and target levels—however, this type is indexed by an implicit argument, **tid**, which represents the type of identifiers for operators. At the source level, this should be thought of as a **string**—the name of the operator in the general purpose language—while at the target level, it is an opaque identifier (implemented as ℕ) that indexes into the DSL's library of operators. This distinction is not frivolous: operators in the source

general-purpose language may be compiled into several operators in the target domain-specific language, and this solution is easier to reason about than name mangling. VERSE currently does not support parametrically polymorphic Coq operators, but compiling them down to ad hoc polymorphism would be another reason to distinguish the representations.

$$
\begin{array}{rcll}
\mathsf{exp}\ \{\mathsf{tid} : \mathsf{Type}\} & ::= & \mathsf{EB}\ (\mathsf{i_b} : \mathsf{bid}) & \text{bound ID} \\
& | & \mathsf{EC}\ (\mathsf{V} : \mathsf{vtype})\ (\mathsf{v} : \mathsf{vraise}\ \mathsf{V}) & \text{constant} \\
& | & \mathsf{EL}\ (\mathsf{i_b} : \mathsf{bid})\ (\mathsf{V} : \mathsf{vtype})\ (\mathsf{e} : \mathsf{exp}) & \text{lambda abstraction} \\
& | & \mathsf{EF}\ (\mathsf{Ts} : \mathsf{ftype})\ (\mathsf{i_t} : \mathsf{tid})\ (\mathsf{es} : \mathsf{exp}^+) & \text{embedded operator}
\end{array}
$$

There are only four cases: a bound variable (the input, or variables introduced by lambda abstractions) named $\mathsf{i_b}$, a constant value v, a lambda abstraction with binder type V, or an application of an embedded operator, indexed by $\mathsf{i_t}$. From here on, for brevity we will omit the implicit **tid** argument in generic definitions. We add a prefix of s or t to instantiate tid as the appropriate source or target level identifier:

$$
\begin{array}{rcl}
\mathsf{sexp} & = & \mathsf{exp\ id} \\
\mathsf{texp} & = & \mathsf{exp\ pid}
\end{array}
$$

Using **exp**, we can define the type of a corpus: it consists of an input type and an output type (specifically, value types), along with a nonempty list of expressions.

$$\mathsf{corpus} = \mathsf{vtype} \times \mathsf{vtype} \times \mathsf{exp}^+$$

5.2.2 Well-Typedness

We are now prepared to discuss VERSE's syntax-driven type system for expressions; we present this somewhat unconventionally before the semantics, because reified types are so essential to the language's execution. We define the following objects:

$$
\begin{array}{rll}
\Delta & : \mathsf{tid} \rightharpoonup (\mathsf{ftype} \times \mathsf{vtype}) & \text{a typing context for operators} \\
\Gamma & : \mathsf{bid} \rightharpoonup \mathsf{vtype} & \text{a typing context for bound variables} \\
\mathsf{i_{b0}} & : \mathsf{bid} & \text{the bound identifier representing the input}
\end{array}
$$

There are three typing judgments, for expressions, argument tuples, and the corpus as a whole:

$$
\begin{array}{rl}
\Delta \mid \Gamma \vdash_{\mathsf{e}} \mathsf{e} : \mathsf{T} & \text{with context } \Delta \text{ and } \Gamma\text{, the expression e has type T} \\
\Delta \mid \Gamma \vdash_{\mathsf{e}^+} \mathsf{es} : \mathsf{Ts} & \text{with context } \Delta \text{ and } \Gamma\text{, the arguments es have ftype Ts} \\
\Delta \vdash_{\mathsf{c}} \mathsf{I}, \mathsf{O}, \mathsf{es} & \text{with context } \Delta\text{, the corpus of programs es has input vtype I and output vtype O}
\end{array}
$$

Their inference rules are defined as follows. We use $\mathsf{i} \mapsto \mathsf{A}$ to represent a binding from i to A present in the typing context. Note that $\vdash_{\mathsf{e}}$ and $\vdash_{\mathsf{e}^+}$ are mutually inductive.

$$\Delta \mid \mathsf{i_b} \mapsto \mathsf{V}, \Gamma \vdash_{\mathsf{e}} (\mathsf{EB}\ \mathsf{i_b}) : \mathsf{TVal}\ \mathsf{V}$$

$$\Delta \mid \Gamma \vdash_{\mathsf{e}} (\mathsf{EC}\ \mathsf{V}\ \mathsf{v}) : \mathsf{TVal}\ \mathsf{V}$$

$$\frac{\Delta \mid \mathsf{i_b} \mapsto \mathsf{V}_1, \Gamma \vdash_{\mathsf{e}} \mathsf{e} : \mathsf{TVal}\ \mathsf{V}_2}{\Delta \mid \Gamma \vdash_{\mathsf{e}} (\mathsf{EL}\ \mathsf{i_b}\ \mathsf{V}_1\ \mathsf{e}) : \mathsf{TFun}\ \mathsf{V}_1\ \mathsf{V}_2}$$

$$\frac{\mathsf{i_t} \mapsto (\mathsf{Ts}, \mathsf{V}), \Delta \mid \Gamma \vdash_{\mathsf{e}^+} \mathsf{es} : \mathsf{Ts}}{\mathsf{i_t} \mapsto (\mathsf{Ts}, \mathsf{V}), \Delta \mid \Gamma \vdash_{\mathsf{e}} (\mathsf{EF}\ \mathsf{Ts}\ \mathsf{i_t}\ \mathsf{es}) : \mathsf{TVal}\ \mathsf{V}}$$

$$\frac{\Delta \mid \Gamma \vdash_{\mathsf{e}} \mathsf{e} : \mathsf{T}}{\Delta \mid \Gamma \vdash_{\mathsf{e}^+} [\mathsf{e}] : [\mathsf{T}]}$$

$$\frac{\Delta \mid \Gamma \vdash_{\mathsf{e}} \mathsf{e} : \mathsf{T} \qquad \Delta \mid \Gamma \vdash_{\mathsf{e}^+} \mathsf{es} : \mathsf{Ts}}{\Delta \mid \Gamma \vdash_{\mathsf{e}^+} (\mathsf{e} :: \mathsf{es}) : (\mathsf{T} :: \mathsf{Ts})}$$

$$\frac{\Delta \mid \mathsf{i_{b0}} \mapsto \mathsf{I} \vdash_{\mathsf{e}} \mathsf{e} : \mathsf{TVal}\ \mathsf{O}}{\Delta \vdash_{\mathsf{c}} (\mathsf{I}, \mathsf{O}, [\mathsf{e}])}$$

$$\frac{\Delta \mid \mathsf{i_{b0}} \mapsto \mathsf{I} \vdash_{\mathsf{e}} \mathsf{e} : \mathsf{TVal}\ \mathsf{O} \qquad \Delta \vdash_{\mathsf{c}} (\mathsf{I}, \mathsf{O}, \mathsf{es})}{\Delta \vdash_{\mathsf{c}} (\mathsf{I}, \mathsf{O}, \mathsf{e} :: \mathsf{es})}$$

Given just these rules, we can already prove some theorems. First, type uniqueness: all well-typed expressions have exactly one type.

Theorem 2 *type uniqueness*

$\forall\ \Delta, \Gamma, e, T_1, T_2.$

$\quad \Delta \mid \Gamma \vdash_e e : T_1 \rightarrow$

$\quad \Delta \mid \Gamma \vdash_e e : T_2 \rightarrow$

$\quad\quad T_1 = T_2$

The proof proceeds by induction on either expressions or the well-typedness judgment.

Typechecking is also decidable; we define a function **tc_corpus** : tid $\rightharpoonup$ (ftype $\times$ vtype) $\rightarrow$ corpus $\rightarrow$ bool, and prove its equivalence to the typing judgments.

Theorem 3 *typechecker correctness*

$\forall\ \Delta, c.\ \ tc\_corpus\ \Delta\ c = true \iff \Delta \vdash_c c$

The proof proceeds by mutual induction on expressions and argument tuples, or on the judgments $\vdash_{\mathsf{e}}$ and $\vdash_{\mathsf{e}^+}$.

5.2.3 Semantics

In order to prove more interesting theorems, we must first define a semantics. We use a large-step, call-by-value environment semantics, with the following equivalents of the typing contexts Δ and Γ:

δ : tid $\rightharpoonup$ (Ts : ftype) $\times$ (V : vtype) $\times$ fsem Ts V	a library of operators
γ : bid $\rightharpoonup$ (V : vtype) $\times$ vraise V	an environment for bound variables

We call the types of Γ and Δ libt and btctxt respectively, and γ and δ lib and bctxt. There are two novelties here. First, we use the notation (a : A) $\times$ B for a dependent pair type, where the type B may be a function of a. Second, we use a new type definition, **fsem** : ftype $\to$ vtype $\to$ Type, which specifies the interface for the forward semantics of an embedded Coq function.

$$\text{fsem Ts V} \;=\; \forall\, \delta, \gamma.\ \text{fraise (traise\_fw exp) Ts} \to (\text{vraise V})^?$$

In words, the forward semantics are a function that takes in a library of operators, a bound variable environment, and a tuple of (optionally) arguments—whose types are raised from Ts—and outputs (optionally) a value whose type is raised from V. For example, if Ts is [TFun TStr TInt; TVal TStr] and V is TInt—that is, if we take as input a function from strings to ints and a string, and return an int—this evaluates to a type of $\forall\, \delta, \gamma.\ (\text{bid} \times \text{vtype} \times \text{exp})^? \times (\text{string})^? \to \mathbb{Z}^?$. A natural operator with this signature is function application, where we apply the first argument to the second and evaluate the resulting expression (using δ and γ). Here we see again the benefit of using option types for the arguments: when the second argument is $\varnothing$, the computation may still succeed if the first argument's function doesn't use its bound variable.

We define two large-step semantic relations, for expressions and argument tuples:

$\langle \delta \mid \gamma \mid \text{e} \rangle \Downarrow_{\text{e}}$ T, t	the expression e evaluates an object t with type raised from T
$\langle \delta \mid \gamma \mid \text{es} \rangle \Downarrow_{\text{e}^+}$ Ts, ts	the arguments es evaluate to a tuple ts with types raised from Ts

The types of t and ts on the right-hand side are somewhat complicated, as they are in a dependent pair with T and Ts respectively. t has type (traise_fw exp T)$^?$ and ts has type fraise (traise_fw exp) Ts. Because these relations are defined for expressions of any type, not just value types, when we want to make statements about the actual final output of a program, we will quantify over value types V and use TVal V as the output type. The inference rules are defined as follows. Note that $\Downarrow_{\text{e}}$ and $\Downarrow_{\text{e}^+}$ are mutually inductive.

$$\langle \delta \mid \text{i}_\text{b} \mapsto (\text{V, v}), \gamma \mid \text{EB i}_\text{b} \rangle \Downarrow_{\text{e}} \text{TVal V, } \lfloor \text{v} \rfloor$$

$$\langle \delta \mid \gamma \mid \text{EC V v} \rangle \Downarrow_{\text{e}} \text{TVal V, } \lfloor \text{v} \rfloor$$

$$\langle \delta \mid \gamma \mid \mathsf{EL}\ \mathsf{i_b}\ \mathsf{V_1}\ \mathsf{e} \rangle \Downarrow_{\mathsf{e}} \mathsf{TFun}\ \mathsf{V_1}\ \mathsf{V_2},\ \lfloor (\mathsf{i_b}, \mathsf{V_1}, \mathsf{e}) \rfloor$$

$$\frac{\langle \mathsf{i_t} \mapsto (\mathsf{Ts}, \mathsf{V}, \mathsf{F}), \delta \mid \gamma \mid \mathsf{es} \rangle \Downarrow_{\mathsf{e}^+} \mathsf{Ts}, \mathsf{ts}}{\langle \mathsf{i_t} \mapsto (\mathsf{Ts}, \mathsf{V}, \mathsf{F}), \delta \mid \gamma \mid \mathsf{EF}\ \mathsf{Ts}\ \mathsf{i_t}\ \mathsf{es} \rangle \Downarrow_{\mathsf{e}} \mathsf{TVal}\ \mathsf{V}, \mathsf{F}\ \mathsf{ts}}$$

$$\frac{\langle \delta \mid \gamma \mid \mathsf{e} \rangle \Downarrow_{\mathsf{e}} \mathsf{T}, \mathsf{t}}{\langle \delta \mid \gamma \mid [\mathsf{e}] \rangle \Downarrow_{\mathsf{e}^+} [\mathsf{T}], \mathsf{t}}$$

$$\frac{\langle \delta \mid \gamma \mid \mathsf{e} \rangle \Downarrow_{\mathsf{e}} \mathsf{T}, \mathsf{t} \qquad \langle \delta \mid \gamma \mid \mathsf{es} \rangle \Downarrow_{\mathsf{e}^+} \mathsf{Ts}, \mathsf{ts}}{\langle \delta \mid \gamma \mid \mathsf{e} :: \mathsf{es} \rangle \Downarrow_{\mathsf{e}^+} (\mathsf{T} :: \mathsf{Ts}), (\mathsf{t}, \mathsf{ts})}$$

Most of these rules are fairly rote, albeit in a dependently typed metalanguage. The most interesting is the rule for EF, where we look up a language operator indexed by $\mathsf{i_t}$ and apply the resulting forward semantics, F, to the arguments ts; this is where embedded Coq functions are integrated into the GPL. Because programs in Coq are guaranteed to terminate, doing this does not negatively impact our ability to reason about language metatheory, even though the computations inside the operators may be arbitrarily complex.

We can now define more formally notions that we used earlier. First, semantic equivalence across expressions of different types: two expressions are equivalent if their input/output behavior is the same.

Definition 7 *semantic equivalence ($e_S \approx e_T$)*
$\forall\ \delta, \gamma.\ \langle \delta \mid \gamma \mid e_S \rangle \Downarrow_e T, t \iff \langle \delta \mid \gamma \mid e_T \rangle \Downarrow_e T, t$

Technically these δ and γ are not the same across different expressions representations—they differ in the type **tid**—but this unnecessarily complicates the statement.

Second, satisfying examples: an expression satisfies an input/output example if it computes this mapping with the large-step semantics. We use $<$ to mean structural membership, e.g. within a list.

Definition 8 *satisfying examples ($\delta \vdash e \vDash xs$)*
$\forall\ i, o, xs.\ (i, o) < xs \to \langle \delta \mid i_{b0} \mapsto (I, i) \mid e \rangle \Downarrow_e O, o$

On top of those definitions, we can now prove type soundness for VERSE's GPL: informally, the statement is that well typed programs will terminate with a value of the correct type. In our formal statement below, we use a relation $\simeq$ that ties typing contexts to environments, essentially stipulating that the type information in the bindings is the same (i.e. we ignore the additional semantic information in δ and γ).

Theorem 4 *type soundness*
$\forall\ \Delta, \Gamma, \delta, \gamma, e, T.$
$\quad \Delta, \Gamma \simeq \delta, \gamma \to$

$$\Delta \mid \Gamma \vdash_e e : T \rightarrow$$
$$\quad \exists\, t.\ \langle \partial \mid \gamma \mid e \rangle \Downarrow_e T, t$$

The proof follows by mutual induction on expressions and argument tuples, or alternately on the definition of $\vdash_e$ and $\vdash_{e^+}$.

We can also prove that the semantics are deterministic, by defining a function **interpret** (p : lib) (I O : vtype) (e : exp) (i : vraise I) : (vraise O)$^{??}$ and proving its equivalence to the large-step semantics. In the double option type returned from this function, the outer layer represents failure due to malformed input (e.g. type errors), while the inner layer represents operator failure within the semantics.

Theorem 5 *interpreter correctness*

$\forall\, \partial, I, O, i, o, e.$
$$\partial \mid i_{b0} \mapsto I \vdash_e e : O \rightarrow$$
$$\quad \textit{interpret}\ \partial\ I\ O\ e\ i = \lfloor o \rfloor \iff \langle \partial \mid i_{b0} \mapsto (I, i) \mid e \rangle \Downarrow_e \textit{TVal}\ O, o$$

Again, the proof follows by mutual induction on $\vdash_e$ and $\vdash_{e^+}$.

5.3 Grammars

The next important kind of object in VERSE is context-free grammars, which are manipulated explicitly in the inference passes and used as a reference in the main deduction pass of synthesis.

5.3.1 Syntax

Again, we share our representation across the source and target level by implicitly parameterizing **tid**. Context-free grammars comprise three inductive types: constructors c, rules r, and DSLs d. These correspond directly to standard forms like BNF, where a rule is a disjunction of constructors and a DSL is a collection of rules rooted at a start symbol.

gconstructor (V : vtype)	::=	GB (i_b : bid)	bound ID
	|	GC (v : vraise V)	specific constant
	|	GT	arbitrary constant
	|	GF (Ts : ftype) (i_t : tid) (is_r: fraiseT rid ft)	operator
grule	::=	GR (V : vtype) (cs : (gconstructor V)$^+$)	normal construtor
	|	GL (i_b : bid) (V_1 : vtype) (i_d : did) (V_2 : vtype)	lambda
dsl	::=	DSL (d : rid $\rightharpoonup$ grule) (ds : did $\rightharpoonup$ dsl)	

There are several differences here when compared with the structure of expressions:

First, there is a new constructor GT for terminal nodes in the grammar that accepts an arbitrary constant. For example, a GT constructor for TStr accepts constant string programs like (EC TStr "foo").

Second, the arguments in GF are represented by a tuple of rids rather than expressions; these specify the production for each argument. Our CFG representation enforces that each constructor consist of at most one embedded operator; nested operators must be spread out into multiple rules. This structure is present in all of our examples of grammars throughout the paper, most prominently in § 4.

Third, we have the addition of GR, which allows us to create a (nonempty) disjunction of constructors. We also separate GL from the rest of the constructors; this is not fundamental, but makes certain aspects of the pipeline easier to write, because constructors are parameterized by a vtype rather than a type.

Finally, we have the dsl type, which is not just a collection of rules, but rather a collection of rules along with a nested collection of DSLs. These nested DSLs are used to represent grammars inside the scope of a lambda abstraction, as seen by the i_d index in GL. This representation dramatically simplifies the generalization pass, because it precludes generalization across binders. For example, generalizing an expression (EB i_{b2}) into a constructor (GB i_{b2}) outside the scope of its original binder may be ill-typed.

5.3.2 Semantics

We will now discuss the "semantics" of our grammars, in the sense that the meaning of a grammar is the sentences it accepts. (This has a cute correspondence with specification by examples: like input/output pairs are examples for the semantics of a program, programs in a corpus are examples for the semantics of a DSL.) We use $<$ to refer to structural membership in a set-like object:

$i_r \mapsto r < d$	the rule index i_r maps to the rule r in the DSL d
$i_d \mapsto d' < d$	the DSL index i_d maps to the recursive DSL d' in the original DSL d
$c < cs$	the constructor c is within the list of constructors cs

In contrast, we use $\in$ to refer to derived membership in a grammar:

$e \in_d d$	the expression e is derivable in DSL d
$e \in_r r \dashv d$	the expression e is derivable from rule r within DSL d
$e \in_c V, c \dashv d$	the expression e is derivable from constructor c of value type V within DSL d
$es \in_{i+} Ts, is_r \dashv d$	the arguments es are derivable from rules indexed by is_r of types Ts within DSL d

These judgments are defined by the following inference rules:

$$\frac{i_{r0} \mapsto r < d \qquad e \in_r r \dashv d}{e \in_d d}$$

$$\frac{c < cs \qquad e \in_c V, c \dashv d}{e \in_r (GR\ V\ cs) \dashv d}$$

$$\frac{i_d \mapsto d' < d \qquad i_{r0} \mapsto r < d' \qquad e' \in_r r \dashv d'}{(EL\ i_b\ V_1\ e') \in_r (GL\ i_b\ V_1\ i_d\ V_2) \dashv d}$$

$$(EB\ i_b) \in_c V, (GB\ i_b) \dashv d$$

$$(EC\ V\ v) \in_c V, (GC\ v) \dashv d$$

$$(EC\ V\ v) \in_c V, GT \dashv d$$

$$\frac{es \in_{i+} Ts, is_r \dashv d}{(EF\ Ts\ i_t\ es) \in_c V, (GF\ Ts\ i_t\ is_r) \dashv d}$$

$$\frac{i_r \mapsto r < d \qquad e \in_r r \dashv d}{[e] \in_{i+} [T], i_r \dashv d}$$

$$\frac{i_r \mapsto r < d \qquad e \in_r r \dashv d \qquad es \in_{i+} Ts, is_r \dashv d}{e :: es \in_{i+} T :: Ts, (i_r, is_r) \dashv d}$$

5.3.3 Well-Typedness

Before we can discuss the well-typedness of DSLs, we have to take a digression back to the type interface for the semantics of language operators. Recall from § 4.1.3 that certain arguments to operators like substring must be finitely enumerable. We define the type **ftbool** : ftype → Type, in order to represent a tuple that picks out which arguments must satisfy this property:

$$ftbool\ Ts \;=\; fraiseT\ bool\ Ts$$

And also **fraiseb** (Ts : ftype) : ftbool Ts → Type, the type of the auxiliary information on fixed arguments that we pass to deduction rules.

$$\begin{aligned} fraiseb\ [T]\ b &= \text{if } b \text{ then } (ttraise\_fw\ T)^? \text{ else unit} \\ fraiseb\ T :: Ts\ (b, bs) &= (\text{if } b \text{ then } (ttraise\_fw\ T)^? \text{ else unit}) \times (fraiseb\ Ts\ bs) \end{aligned}$$

Using these, we can define the interface for backward semantics (deduction rules) called **bsem** : ∀ (Ts : ftype), vtype → ftbool Ts → Type.

$$bsem\ Ts\ V\ bs \;=\; vraise\ V \to fraiseb\ Ts\ bs \to (fraise\_bw\ Ts)^*$$

For example, consider the substring operator, where Ts is [TVal TStr; TVal TInt; TVal TInt], V is TStr, and bs is (true, false, false). The deduction rule must therefore have the mathematical type string $\to$ (string$^?$ $\times$ unit $\times$ unit) $\to$ (string$^?$ $\times$ $\mathbb{Z}^?$ $\times$ $\mathbb{Z}^?$)*. That is to say, the deduction rule receives the output string and the first argument as input (the second and third unit arguments are vacuous), and must return a list of possible input tuples. Now we can finally define the type of an embedded operator for the standard library:

```
op ::= OP (Ts : ftype) (V : vtype) (bs : ftbool ft) (F : fsem exp Ts V) (B : bsem Ts V bs) (P : equiv Ts V bs F B)
```

P is a proof of the correspondence between the forward semantics F and backward semantics B.

Now we return to defining well-typedness for DSLs, which will include conditions on well-foundedness for sub-DSLs rooted at rules marked as finitely enumerable. We use the following judgments, where Δ is now augmented with ftbools for each operator.

$\Delta \mid \Gamma \vdash_{\mathsf{d}} \mathsf{d} : \mathsf{V}$	with context Δ and Γ, the DSL d has value type V
$\Delta \mid \Gamma \vdash_{\mathsf{r}} \mathsf{r} : \mathsf{T} \dashv \mathsf{d}$	with context Δ and Γ, the rule r has type T in DSL d
$\Delta \mid \Gamma \vdash_{\mathsf{cs}} \mathsf{cs} : \mathsf{V} \dashv \mathsf{d}$	with context Δ and Γ, the constructors cs have value type V in DSL d
$\Delta \mid \Gamma \vdash_{\mathsf{c}} \mathsf{c} : \mathsf{V} \dashv \mathsf{d}$	with context Δ and Γ, the constructor c has value type V in DSL d
$\Delta \mid \Gamma \vdash_{\mathsf{i}+} \mathsf{is_r} : \mathsf{Ts}, \mathsf{bs} \dashv \mathsf{d}$	with context Δ and Γ, the rules indexed by $\mathsf{is_r}$ have types Ts and ftbools bs in DSL d
$\mathsf{d} \Downarrow_{\mathsf{d}}$	the DSL d is well-founded (i.e. finitely enumerable)
$\mathsf{r} \Downarrow_{\mathsf{r}} \dashv \mathsf{d}$	the rule r is well-founded in DSL d
$\mathsf{V}, \mathsf{cs} \Downarrow_{\mathsf{cs}} \dashv \mathsf{d}$	the constructors cs of value type V are well-founded in DSL d
$\mathsf{V}, \mathsf{c} \Downarrow_{\mathsf{c}} \dashv \mathsf{d}$	the constructor c of value type V is well-founded in DSL d
$\mathsf{Ts}, \mathsf{is_r} \Downarrow_{\mathsf{i}+} \dashv \mathsf{d}$	the rules indexed by $\mathsf{is_r}$ of types Ts are well-founded in DSL d

When defining the well-typedness inference rules, we have to take care to ensure that the judgment can actually be derived with finite proof trees. Our previous inference rules (and those imminently for well-foundedness) are based on the finite depth of an expression, whether explicitly or as a path through the grammar, so they would fail to be effective on DSLs with an infinite number of derivable programs (which have loop in the grammar). One approach that we could use to resolve this is making the well-typedness judgment coinductive, but Coq's built-in support for coinduction is notoriously inflexible [38]. Instead, we will check whether a rule is well-typed conditioned on the other rules' type annotations being correct, and then require that this simultaneously hold of all rules. See the key inference rules defined below; most are too similar to prior rules to be worth repeating.

$$\frac{\mathsf{i_{r0}} \mapsto (\mathsf{GR}\ \mathsf{V}\ \mathsf{cs}) < \mathsf{d} \qquad \forall\, \mathsf{i_r}, \mathsf{r}.\ \ \mathsf{i_r} \mapsto \mathsf{r} < \mathsf{d} \to \exists\, \mathsf{T}.\ \ \Delta \mid \Gamma \vdash_{\mathsf{r}} \mathsf{r} : \mathsf{T} \dashv \mathsf{d}}{\Delta \mid \Gamma \vdash_{\mathsf{d}} \mathsf{d} : \mathsf{V}}$$

The right premise here is critical: it requires that all rules in the DSL be locally well-typed.

$$\frac{\mathsf{i_r} \mapsto (\mathsf{GR\ V\ cs}) < \mathsf{d} \qquad \mathsf{b} = \mathsf{true} \to (\mathsf{GR\ V\ cs}) \Downarrow_{\mathsf{r}} \dashv \mathsf{d}}{\Delta \mid \Gamma \vdash_{\mathsf{i}^+} \mathsf{i_r} : [\mathsf{TVal\ V}], \mathsf{b} \dashv \mathsf{d}}$$

$$\frac{\mathsf{i_r} \mapsto (\mathsf{GR\ V\ cs}) < \mathsf{d} \qquad \mathsf{b} = \mathsf{true} \to (\mathsf{GL\ i_b\ V_1\ i_d\ V_2}) \Downarrow_{\mathsf{r}} \dashv \mathsf{d}}{\Delta \mid \Gamma \vdash_{\mathsf{i}^+} \mathsf{i_r} : [\mathsf{TFun\ V_1\ V_2}], \mathsf{b} \dashv \mathsf{d}}$$

These inference rules are where the conditioning happens: there is no $\vdash_{\mathsf{r}}$ premise, but rather just a check that the reified types in the rule match what they are supposed to. When an argument is marked for finite enumeration by b, it also requires that the rule be well-founded.

$$\mathsf{V}, (\mathsf{GB\ i_b}) \Downarrow_{\mathsf{c}} \dashv \mathsf{d}$$

$$\mathsf{V}, (\mathsf{GC\ v}) \Downarrow_{\mathsf{c}} \dashv \mathsf{d}$$

$$\mathsf{TBool}, \mathsf{GT} \Downarrow_{\mathsf{c}} \dashv \mathsf{d}$$

$$\frac{\mathsf{V_1}, \mathsf{GT} \Downarrow_{\mathsf{c}} \dashv \mathsf{d} \qquad \mathsf{V_2}, \mathsf{GT} \Downarrow_{\mathsf{c}} \dashv \mathsf{d}}{\mathsf{TPair\ V_1\ V_2}, \mathsf{GT} \Downarrow_{\mathsf{c}} \dashv \mathsf{d}}$$

$$\frac{\mathsf{Ts}, \mathsf{is_r} \Downarrow_{\mathsf{i}^+} \dashv \mathsf{d}}{\mathsf{V}, (\mathsf{GF\ Ts\ i_t\ is_r}) \Downarrow_{\mathsf{c}} \dashv \mathsf{d}}$$

The most interesting part of the well-foundedness judgment lies in the constructor: bound variables, constants, and operators can be finitely enumerated, but so can GT terminals for types with finite universes, like TBool and tuples of them constructed with TPair.

Given these typing rules, we can prove a higher-order equivalent of type soundness for grammars: roughly, if a DSL has type T, every program in the DSL has type T.

Theorem 6 *grammar type soundness*

$\forall\, \Delta, \Gamma, d, e, O.$

$\quad \Delta \mid \Gamma \vdash_d d : O \to$

$\quad e \in_d d \to$

$\quad\quad \Delta \mid \Gamma \vdash_e e : \mathit{TVal}\ O$

The proof can be structured as a mutual recursion on three lemmas: the soundness of $\vdash_{\mathsf{c}}$, $\vdash_{\mathsf{i}^+}$, and $\vdash_{\mathsf{r}}$, given the well-typedness of the containing DSL as a premise. Note that this is not a straightforward

mutual induction: only $\vdash_c$ and $\vdash_{i+}$ are defined in a mutually inductive way, in order to make proof trees finite. The mutual recursion of the proof can be shown to be well-founded by measuring the size of the expression e being derived. It may be possible to mutually induct on the definitions of $\in_r$, $\in_{i+}$, and $\in_c$ more straightforwardly, as they are defined together.

Typechecking is again decidable; we define a function **tc_dsl** : libt → btctxt → dsl → vtype → bool, which validates that i_{r0} has type TVal V and that every rule in the grammar is well-typed conditioned on the annotations of other rules, and then prove its equivalence to the typing judgments.

Theorem 7 *grammar typechecker correctness*
$\forall\ \Delta, \Gamma, d, V.\ \ tc\_dsl\ \Delta\ \Gamma\ d\ V = true \iff \Delta \mid \Gamma \vdash_d d : V$

The proof proceeds largely by mutual induction on the definitions of $\vdash_c$ and $\vdash_{i+}$, with an additional induction on the number of rules in the DSL for the property that every rule is locally well-typed.

5.4 Data Structures

At last, we have reached the data structures used for VERSE's deductive synthesis: version space algebras (VSAs) (§ 5.4.1), intersected VSAs (§ 5.4.2), and program spaces (§ 5.4.3).

5.4.1 Version Space Algebras

VSAs, described informally in Section 4.1.3, have similar structure to DSLs above.

vconstructor (V : vtype)	::=	VB (i_b : bid)	bound ID
	|	VC (v : vraise V)	specific constant
	|	VF (Ts : ftype) (i_p : pid) (is_{v+r} : fraiseT (vid + rid) ft)	operator
vrule	::=	VR (V : vtype) (cs : (vconstructor V)*)	normal constructor
	|	VL (i_b : bid) (V_1 : vtype) (i_s : sid) (V_2 : vtype)	lambda
	|	VU	waiting in the queue for BFS

First, let's motivate the differences between gconstructor & vconstructor and grule & vrule.

vconstructor has no constructor VT; this is because VSAs represent a set of concrete expressions, rather than acceptance criteria for those expressions. When a GT constructor is reached during deductive synthesis, it will create a VC constructor with the constant necessitated by the output specification. For example, trying to synthesize a program mapping "foo" to "bar" in a DSL with only a GT rule for strings will result in a VSA with just the constructor VC "bar".

It also uses the sum type vid + rid, rather than just vid or rid alone, to represent the location of an argument's production rule. Most of the time, the left injection will be used here, storing a vid representing the set of programs that output the deduced value. The right injection, rid, is for situations like if then else, where some of the inputs are deduced to be unconstrained. Here, the right thing to do to achieve complete synthesis is to generate all valid programs in the original DSL (where the production rule is the stored rid), when performing bottom-up enumeration later.

vrule most notably has a new constructor, VU, which represents a rule+value pair that is waiting in the queue for the next iteration of breadth first search. This is not strictly necessary for correctness, but its presence makes it easier to state invariants about the VSA during synthesis. It also now allows empty lists of vconstructors in VR: this is because deduction only suggests possible inputs, not necessarily ones that can actually be derived with the forward semantics of the DSL—whereas a CFG rule with no constructors is nonsensical.

Now we present the definition of vsa in terms of vconstructor and vrule. We leave the types mapping and queue opaque for now; they represent mappings from DSL rules+values to VSA rules and the BFS queue of VSA rules respectively, for internal use during iterated deductive synthesis.

$$\mathsf{vsa} \ ::= \ \mathsf{VSA}\ (\mathsf{a} : \mathsf{vid} \rightharpoonup \mathsf{vrule})\ (\mathsf{as} : \mathsf{sid} \rightharpoonup (\mathsf{bctxt} \times \mathsf{vsa} \times \mathsf{mapping} \times \mathsf{queue})^{+})\ (\mathsf{i_v}\colon \mathsf{vid})\ (\mathsf{i_s}\colon \mathsf{sid})$$

The most cryptic part of this definition is the second parameter, as, which stores the necessary state to resume recursive synthesis for lambda functions, indexed by a state identifier $\mathsf{i_s}$. This is a nonempty list because there is a VSA for each example pair; once the VSAs have been intersected in the next pass, this will collapse to a single IVSA. $\mathsf{i_v}$ and $\mathsf{i_s}$ are running maximum indices used to allocate fresh identifiers.

We could define relations for membership of expressions in the VSA, $\mathsf{e} \in_{\mathsf{a}} \mathsf{a} \dashv \mathsf{d}$, and well-typedness of the VSA, $\Delta \mid \Gamma \vdash_{\mathsf{a}} \mathsf{a} : \mathsf{V} \dashv \mathsf{d}$. By this point, I expect the reader has grown weary of inference rules—the author certainly has—so we provide the key rules dealing with the vid + rid sum type.

$$\frac{\mathsf{i_v} \mapsto \mathsf{vr} < \mathsf{a} \qquad \mathsf{e} \in_{\mathsf{vr}} \mathsf{vr} \dashv \mathsf{a}, \mathsf{d}}{[\mathsf{e}] \in_{\mathsf{i}^+} [\mathsf{T}], \mathsf{inl}\ \mathsf{i_v} \dashv \mathsf{a}, \mathsf{d}}$$

$$\frac{\mathsf{i_r} \mapsto \mathsf{r} < \mathsf{d} \qquad \mathsf{e} \in_{\mathsf{r}} \mathsf{r} \dashv \mathsf{d}}{[\mathsf{e}] \in_{\mathsf{i}^+} [\mathsf{T}], \mathsf{inr}\ \mathsf{i_r} \dashv \mathsf{a}, \mathsf{d}}$$

5.4.2 Intersected Version Space Algebras

Intersected VSAs use the same vconstructor and vrule definitions as plain VSAs, bleaching away the state required to continue synthesis at a later time and intersecting the recursive VSAs for synthesis of lambda functions.

$$\text{ivsa} \;::=\; \text{IVSA}\,(\text{ia} : \text{vid} \rightharpoonup \text{vrule})\,(\text{ias} : \text{sid} \rightharpoonup \text{ivsa})$$

Again, one can straightforwardly define sound membership and well-typedness judgments.

5.4.3 Program Spaces

Program spaces share the same inductive structure as vsas and ivsas, but they are defined directly in terms of expressions rather than concisely like VSAs.

$$\text{pspace} \;::=\; \text{PS}\,(\text{s} : \text{vid} \rightharpoonup \text{texp})\,(\text{ss} : \text{sid} \rightharpoonup \text{pspace})$$

Each of these sets contains the programs derivable from the associated production rule in the VSA; in the case of infinite program spaces, these sets become increasingly better approximations as we enumerate more steps. Membership and well-typedness are trivial here: membership is just set membership in the canonical root $\text{i}_{\text{v}0}$, while well-typedness is the conjunction of well-typedness for every expression within the space.

6. Inference

In this section, we will provide proof sketches for the inference passes of VERSE. We will overload Δ and Γ to represent several kinds of contexts simultaneously in order to avoid clutter. Recall from Section 4.2 the high-level guarantee that we want from the inference engine: the resulting DSL should semantically preserve membership of the programs in the corpus. We will phrase this in terms of a function **infer** : slib $\to$ scorpus $\to$ tdsl$^?$. (The option return type is only used for malformed inputs.)

Theorem 8 *corpus semantic membership preservation*

$$\begin{aligned}
&\forall\, \Delta\; I\; O\; es_S.\\
&\quad \Delta \vdash_c I, O, es_S \to\\
&\qquad \exists\, \boldsymbol{d_T}.\ \mathit{infer}\ \Delta\ (I, O, es_S) = \lfloor \boldsymbol{d_T} \rfloor\\
&\qquad\quad \wedge\ \Delta \mid i_{b0} \mapsto I \vdash_d \boldsymbol{d_T} : O\\
&\qquad\quad \wedge\ \forall\, e_S.\ e_S < es_S \to \exists\, \boldsymbol{e_T}.\ e_S \approx \boldsymbol{e_T}\\
&\qquad\qquad\qquad\qquad \wedge\ \boldsymbol{e_T} \in_d \boldsymbol{d_T}
\end{aligned}$$

In words, if the corpus is well-typed, infer will output a well-typed DSL that semantically preserves membership of programs in the corpus.

Before we dive into the details of individual passes, a note on proof strategy: we aim to show that the inference algorithm will work correctly on well-formed input by proving properties on the translation passes themselves. An alternative approach called *translation validation*, used occasionally by projects

like CompCert [47], treats translators as unverified, instead running the output through a verified validator. This results in only a formal guarantee of soundness—the unverified translator may be incorrect, causing the validator to reject the output and fail—but this is much easier to mechanize than a traditional correctness proof. In the context of VERSE's inference algorithm, this validator would check the membership of the original corpus programs in the final DSL, and for the synthesis engine, it would check that each synthesized program satisfies the input/output examples before outputting them. We are interested in proving the completeness of VERSE, i.e. that it will succeed given well-formed input, so we do not take this approach.

6.1 Embedding

The embedding pass consists of a function **embed** : slib $\to$ scorpus $\to$ sdsl$^?$, where the returned DSL is a disjunction of the programs in the corpus. The correctness of this step can be proven using the following stronger lemma, which states that syntactic membership of programs in the corpus is preserved when we translate into a DSL.

Lemma 1 *corpus syntactic membership preservation*

$$\begin{aligned}
&\forall\, \Delta\ I\ O\ es_S.\\
&\quad \Delta \vdash_c I, O, es_S \to\\
&\qquad \exists\, d_T.\ \mathit{embed}\ \Delta\ (I, O, es_S) = \lfloor d_T \rfloor\\
&\qquad\quad \wedge\ \Delta \mid i_{b0} \mapsto I \vdash_d d_T : O\\
&\qquad\quad \wedge\ \forall\, e.\ e < es_S \to e \in_d d_T
\end{aligned}$$

This follows from induction on the length of the corpus, provided lemmas about helper functions that embed expressions & argument tuples and an invariant about the ability to generate fresh identifiers. These lemmas will be proven by mutual induction on expressions & argument tuples (or their typing judgments).

For an example of fresh identifiers, our implementation instantiates **rid** as $\mathbb{N}$ and keeps a running maximum index used so far, incrementing to allocate a new one. Given this maximum n, the natural invariant is $\forall\, i_r.\ i_r \geq n \to \neg\exists\, r.\ i_r \mapsto r < d$.

Corpus semantic membership preservation follows cleanly from corpus syntactic membership preservation: the existential e_T can be constructed from e_S with the identity function.

6.2 Semantification

The semantification pass consists of a function **semantify** : slib $\to$ sdsl $\to$ tlib $\times$ tdsl, where the returned DSL indexes its operators with **pid** rather than **id**, and operators used in rules that must be well-founded

have different aliases in the library from those that need not be. For example, the append in substring(append(_, _), _, _) will be translated differently from just append(_, _). We again use a stronger lemma, which states that every program in the source language is semantically preserved, not just those in the corpus—though it happens to be the case in VERSE's pipeline that these coincide.

Lemma 2 *semantic membership preservation*

$$
\begin{aligned}
&\forall\ \Delta_S\ \Gamma\ d_S\ V.\\
&\quad \Delta_S \mid \Gamma \vdash_d d_S : V \rightarrow\\
&\qquad \exists\ \Delta_T\ \boldsymbol{d_T}.\ \mathit{semantify}\ \Delta_S\ d_S = (\Delta_T, \boldsymbol{d_T})\\
&\qquad\qquad \wedge\ \Delta_S \simeq \Delta_T\\
&\qquad\qquad \wedge\ \Delta_T \mid \Gamma \vdash_d \boldsymbol{d_T} : V\\
&\qquad\qquad \wedge\ \forall\ e_S.\ e_S \in_d d_S \rightarrow \exists\ \boldsymbol{e_T}.\ e_S \approx \boldsymbol{e_T}\\
&\qquad\qquad\qquad\qquad \wedge\ \boldsymbol{e_T} \in_d \boldsymbol{d_T}
\end{aligned}
$$

The proof follows by induction on expressions. Every case except EF is trivial, because they are translated syntactically. Again we need to keep an invariant for fresh identifier generation, though this can be regenerated independently in each pass by folding over the collection of operators to find the maximum.

Semantic membership preservation can be chained with the proof for the embedding pass by transitivity, getting closer to end-to-end corpus semantic membership preservation.

6.3 Generalization

The generalization pass consists of a function **generalize** : tlib → tdsl → tdsl, which broadens the input DSL in the way specified in § 4.1.2—roughly, by using one grammar rule for each argument of an operator, wherever it appears. Here, the membership of programs in the language is syntactically preserved.

Lemma 3 *syntactic membership preservation*

$$
\begin{aligned}
&\forall\ \Delta\ \Gamma\ V\ d_S.\\
&\quad \Delta \mid \Gamma \vdash_d d_S : V \rightarrow\\
&\qquad \exists\ \boldsymbol{d'_T}.\ \mathit{generalize}\ \Delta\ d_S = \boldsymbol{d'_T}\\
&\qquad\qquad \wedge\ \Delta \mid \Gamma \vdash_d \boldsymbol{d'_T} : V\\
&\qquad\qquad \wedge\ \forall\ e.\ e\ \in_d\ d_S \rightarrow e\ \in_d\ \boldsymbol{d'_T}
\end{aligned}
$$

The proof of this transformation depends on its implementation: if the pass consists of repeated, local transformations, we can prove each of these correct and the entirety by induction. If the pass creates mappings by folding over the library of operators, and then transforms the DSL using these mappings, it

makes more sense to induct on the library of operators, and then on the traversal of the DSL. Either way, the key lemma is that for each old rule identifier replaced with a new rule identifier, the new identifier points to a rule that has at least the same constructors as the old one.

We can chain syntactic membership preservation with the proof from the semantification pass by transitivity, giving us the full corpus semantic membership preservation result.

7. Synthesis

In this section, we will provide proof sketches for the synthesis passes of VERSE. Recall from Section 4.2 the high-level guarantee we want from the synthesis engine: given a set of example input/output pairs, we should output exactly those programs in the DSL that satisfy the example pairs. When we allow infinite program spaces, this becomes a guarantee that any particular program satisfying the example pairs will eventually be output. We phrase this formally below, in terms of a function **synthesize** : $\forall$ (I O : vtype). tlib $\to$ tdsl $\to$ examples I O $\to \mathbb{N} \to$ (list texp)$^?$. The natural number argument designates the number of iterations of BFS that should be performed; this pattern, where a finite amount of "fuel" is used to ensure that a function terminates, is common in Coq.

Theorem 9 *synthesizer correctness*

$\forall\ \Delta, I, O, d, xs, e.$

$\quad \Delta \mid i_{b0} \mapsto I \vdash_d d : O \to$

$\quad\quad \exists\ n, es.\ \mathit{synthesize}\ \Delta\ d\ xs\ n = \lfloor es \rfloor$

$\quad\quad\quad \wedge\ \Delta \vdash e \vDash xs \iff e < es$

In words, this says that if the DSL is well-typed, for every expression e there exists a fuel value such that the domain-specialized synthesizer will output e if and only if it satisfies the example specification.

7.1 Deduction

The deduction pass consists of a function **deduce** : $\forall$ (I O : vtype). tlib $\to$ tdsl $\to$ examples I O $\to \mathbb{N} \to$ (vsa$^+$)$^?$, which computes n iterations of top-down deductive synthesis on each example pair. We will talk only about deduction on a single example, using **deduce1** : $\forall$ (I O : vtype). tlib $\to$ tdsl $\to$ example I O $\to \mathbb{N} \to$ vsa$^?$, without loss of generality. The correctness statement for **deduce1** says that exactly the expressions satisfying the given example should be derivable in the VSA, given enough time. This is similar to the overall synthesis correctness statement, but for only one example and with VSA membership rather than list membership.

Lemma 4 *deduction correctness*

$$\forall\, \Delta, I, O, d, x, e.$$
$$\Delta \mid i_{b0} \mapsto I \vdash_d d : O \to$$
$$\exists\, n, a.\ \mathit{deduce1}\ \Delta\ d\ x\ n = \lfloor a \rfloor$$
$$\wedge\ \Delta \vdash e \vDash x \iff e \in_a a \dashv d$$

In words, if the DSL is well-typed, for every expression e there is a fuel value such that the deduced VSA will contain e if and only if it satisfies the input/output example x.

One needs a lot of machinery to prove this formally: the BFS queue and mapping connecting the DSL and VSA need a state invariant and the direction of enumeration is opposite the direction that expressions are defined inductively (i.e. we fill in holes for subexpressions over time). But the lemma at the heart of the proof is the correspondence between forward and backward semantics, given by the interface for embedded Coq functions. We define this correspondence below:

Definition 9 *forward/backward semantics correspondence (*equiv Ts V bs F B*)*

$$\forall\, \delta, \gamma, is, o, ps.\ (F\ \delta\ \gamma\ is) = \lfloor o \rfloor \iff Ts \mid bs \mid ps \mid is \rhd_{is} (B\ o\ ps)$$

That is to say, the forward semantics successfully evaluates an input argument tuple to a value if and only if there is some argument tuple deduced by the backward semantics that satisfies the Ts | bs | ps | is▷B o ps relation. For clarity, Ts is an ftype, bs is an ftbool Ts, ps is an fraiseb Ts bs, is is an fraise traise_fw Ts, and (B o ps) is a (fraise_bw Ts)*. We will describe this relation in words, rather than formally, to avoid dependent type hell.

First, for each b in bs that is true—i.e. where the argument will be finitely enumerated and provided to the deduction rule—every output of (B o ps) must use return the corresponding p. That is to say, deduction rules have to abide by the fixed parameters they are given. This handles deduction with finite enumeration, liks substring.

Second, there must exist a deduced argument tuple in (B o ps) where the arguments are either identical to is, or are ∅ when is have concrete values. This handles unconstrained deduction, like ifthenelse.

Third, for function types, where traise_fw and traise_bw behave differently, we require instead that the function i within the input argument tuple is satisfy the input/output mappings specified by the corresponding deduced argument. This handles cases like apply and map.

These properties allow us to derive expressions that satisfy the specified example under the forward semantics, while only using the backward semantics to traverse the search space.

7.2 Intersection

The intersection pass consists of a function **intersect** : $\mathsf{vsa}^+ \to \mathsf{ivsa}^?$, which conceptually intersects the sets of programs represented by the input VSAs, as described in Section 4.1.3. The option return type is only for malformed inputs; intersection should always succeed on well-typed input VSAs. It is easier to think about intersection as an operation on two VSAs, **intersect2** : $\mathsf{vsa} \to \mathsf{vsa} \to \mathsf{ivsa}^?$, which we can fold over the entire list. Given the correctness of intersection for two VSAs, we can extend this to the entire list by induction.

Lemma 5 *intersection correctness*

$$
\begin{aligned}
&\forall\, \Delta, \Gamma, a_1, a_2, V.\\
&\quad \Delta \mid \Gamma \vdash_a a_1 : V \dashv d \to\\
&\quad \Delta \mid \Gamma \vdash_a a_2 : V \dashv d \to\\
&\qquad \exists\, ia.\ \mathit{intersection2}\ a_1\ a_2 = \lfloor ia \rfloor\\
&\qquad\quad \wedge\ \Delta \mid \Gamma \vdash_{ia} ia : V \dashv d\\
&\qquad\quad \wedge\ \forall\, e.\ e \in_{ia} ia \dashv d \iff (e \in_a a_1 \dashv d \wedge e \in_a a_2 \dashv d)
\end{aligned}
$$

In words, for two well-typed VSAs, an expression should be present in the intersected VSA if and only if it was present in both of the original VSAs.

The proof of this is difficult in the setting where VSAs can have cycles (i.e. can have infinite program space), because the traversal structure doesn't correspond to any natural inductive structure; we have to keep a set of visited notes to avoid infinite loops. When we combine this with *deduction correctness* above, we get that the resulting ia satisfies both a_1 and a_2's examples.

7.3 Enumeration

The enumeration pass consists of a function **enumerate** : $\mathsf{ivsa} \to \mathbb{N} \to \mathsf{pspace}$, which performs bottom-up enumeration as described in Section 4.1.3. The correctness statement requires that exactly the programs in the intersected VSA must be generated, given enough time.

Lemma 6 *enumeration correctness*

$$
\begin{aligned}
&\forall\, \Delta, \Gamma, ia, V, e.\\
&\quad \Delta \mid \Gamma \vdash_{ia} ia : V \dashv d \to\\
&\qquad \exists\, n, s.\ \mathit{enumerate}\ ia\ n = s\\
&\qquad\quad \wedge\ e \in_s s \iff e \in_{ia} ia \dashv d
\end{aligned}
$$

In words, if an intersected VSA is well-typed, for every expression e there is a fuel value such that bottom-up enumeration will output e if and only if it a member of the VSA. The proof here is simpler than for top-down synthesis, because we are building up expressions from subexpressions first, in the same way that they are defined inductively, and without any care for semantics—the process is entirely syntactic. Define a variant of the $\in$ relation that includes the number of steps, as $e \in_{vc} V$, $vr \dashv d \uparrow n$, where n is the depth of the expression being derived. Terminal nodes (bound IDs and constants) can be derived in 0 steps, while nonterminal nodes (embedded operators) can be derived in a number of steps that is the maximum of all their arguments, plus 1. Now it suffices to prove that bottom-up enumeration contains all n-derivable expressions after n steps, and that this augmented relation is equivalent to the original one.

7.4 Output

The output pass consists of a function **output** : pspace $\to$ texp*, which returns the set of texps indexed by i_{v0} in the pspace. The proof of correctness here is trivial, given the correctness of the enumeration pass and the definition of $\in_s$, which is rooted at i_{v0}. With this simple step, we reach the end of VERSE's synthesis engine correctness proof. The inference and synthesis correctness theorems compose naturally into VERSE's formal guarantee for developers: that—provided a well-typed corpus—VERSE will output a well-typed DSL that semantically preserves the programs in the corpus, as well as a synthesizer that is both sound and complete with respect to the DSL.

Theorem 10 *VERSE correctness*

$$
\begin{aligned}
&\forall\, \Delta, I, O, es_S.\\
&\quad \Delta \vdash_c I, O, es_S \to\\
&\qquad \exists\, d_T.\ \mathit{VERSE}\ \Delta\ (I, O, es_S) = \lfloor (d_T, \mathit{synthesize}\ \Delta\ d_T) \rfloor\\
&\qquad\quad \wedge\ \Delta \mid i_{b0} \mapsto I \vdash_d d_T : O\\
&\qquad\quad \wedge\ \forall\, e_S.\ e_S < es_S \to \exists\, e_T.\ e_S \approx e_T\\
&\qquad\qquad\qquad\qquad \wedge\ e_T \in_d d_T\\
&\qquad\quad \wedge\ \forall\, e_T, xs.\ \exists\, n, es_T.\ \mathit{synthesize}\ \Delta\ d_T\ xs\ n = \lfloor es_T \rfloor\\
&\qquad\qquad\qquad\qquad \wedge\ \Delta \vdash e_T \vDash xs \iff e_T < es_T
\end{aligned}
$$

8. Implementation

In this section, we make explicit the assumptions (§ 8.1) and challenges (§ 8.2) of our Coq development. The VERSE project is publicly available and under active development at https://github.com/gtanzer/verse. We currently have most of the metatheory from (§ 5) mechanized, but we do not yet have complete proofs

of correctness for the passes of VERSE itself—only some nontrivial proofs of termination required for Coq to accept the programs at all. Due to recent changes in some type definitions—most notably dsl—the synthesis engine does not compile on the main branch; we have available a secondary branch with an older, working version. These type definitions changes would not normally be so catastrophic, as they change very little of the synthesis engine code (only the lambda cases), but they also break the auxiliary functions and proofs we need in order to prove termination.

8.1 Assumptions

Our trusted computing base is the Coq kernel, which consists of approximately 14,000 lines of OCaml code [3]. This, of course, means that an OCaml compiler or interpreter must also be trusted. If we extract VERSE to OCaml, we additionally assume the correctness of Coq's extraction mechanism. There are efforts underway to remove some of these assumptions, such as the MetaCoq project to formalize Coq in Coq [73], Sozeau et al.'s verified typechecker for Coq [74], and Müllen et al.'s Œuf [53] & Anand et al.'s CertiCoq [15], which verify extraction from Coq to assembly. We do not use these components, as they are not yet integrated into the standard Coq release.

We additionally use two axioms: soundness of heterogeneous equality [51] and Streicher's axiom K [77]. The former states that heterogeneous equality, i.e. equality parameterized by two types, where the objects can only be equal if the types are equal, implies homogeneous equality. The latter states that all proofs of the same reflexive equality, i.e. of propositions with the form `x = x`, are equal. (This is a weaker form of proof irrelevance, which states that all proofs of the same proposition are equal.) These axioms, which are known to be independent but consistent with Coq's logic, are dependencies of the `Program.Equality` library, which augments common Coq tactics to work with dependent types. They are probably not necessary for our development, but they provide a substantial quality of life improvement when writing proofs. Many works that verify systems using dependent types, such as CompCert [47], admit these axioms (among others).

8.2 Challenges

There are three main areas in which our presentation in the sections above and our Coq development diverge: mutual recursion, mutual induction, and finite maps. These details would have unnecessarily complicated the description of VERSE, but we hope that they are helpful to others facing the same obstacles, or those who wish to understand the actual source code.

8.2.1 Mutual Recursion

Coq's `Program Fixpoint` vernacular command, which makes it easier to write termination proofs for recursive functions, doesn't yet support mutual recursion. So, although we describe several functions above as plain mutual recursion, we concretely implement them as a single `Fixpoint`; we strongly recommend this compared to the alternatives, such as manually passing a structurally decreasing relation between mutually recursive functions. See an example of this pattern in Figure 8.1. Although better than the alternatives, the workflow is still unpleasant: we typically first write the program with normal mutual recursion, then manually translate it into its unreadable final form.

8.2.2 Mutual Induction

A natural way to write mutually inductive proofs is to state the lemmas as mutually recursive, and use each of them as hypotheses in the proofs of the others (while taking care that this is done in a well-founded way). Structuring proofs in this way is supported by Coq, but once their translated proof terms become moderately large, `Qed` starts to hang. We can avoid this problem by manually applying mutual induction principles generated with the `Combined Scheme` vernacular command. See an example of this pattern in Figure 8.2.

However, `Combined Scheme` only generates mutual induction principles for types that are defined with mutual induction; we may want to structure other proofs, such as the proof of DSL type soundness in § 5.3.3, in a well-founded mutual recursion that does not correspond directly to the type definitions. This is the proof equivalent of our problem in § 8.2.1 with non-syntactic mutual recursion for programs; we need to prove that our proof terminates. But if we try to solve this problem in the same way—by transforming our mutually recursive lemmas into a singly recursive form—we find that `Program Fixpoint` doesn't support definitions through Coq's tactic language Ltac, which is how virtually all proof terms are written. This is in contrast to standard `Fixpoint`, where tactics can be used to prove that the return type is inhabited by constructing it, either for propositions or types with computational content. We have found this feature (interactive programming with tactics) quite helpful for writing programs that make extensive use of dependent types, since the alternative is verbose type annotations like the so-called convoy pattern [2]. But we know of no good way to prove well-foundedness for functions or proofs written in this way. The low-level fallback of defining a structurally decreasing relation, which is the method that `Program Fixpoint` targets in its translation, is prohibitively time-consuming.

```
Fixpoint P (a : A) (b : B) {???} : C := ...
with Q (d : D) (e : E) {???} : F := ...
```

```
Inductive args : Type :=
  | P (a : A) (b : B)
  | Q (d : D) (e : E).

Definition ret (x : args) : Type :=
  match x with
  | P _ _ => C
  | Q _ _ => F
  end.

Fixpoint countP (a : A) (b : B) : nat := ...
    with countQ (d : D) (e : E) : nat := ...

Definition count (x : args) : nat :=
  match x with
  | P a b => countP a b
  | Q d e => countQ d e
  end.

Program Fixpoint PQ (x : args) {measure (count x)}: ret x :=
  match x with
  | P a b => ...
  | Q d e => ...
  end.
```

Figure 8.1: An example of the pattern we use to prove termination of mutually recursive functions using `Program Fixpoint`. The top is the original and the bottom is the transformed version.

```
Lemma A :
  forall (t : T1), ...
with B :
  forall (t : T2), ...
Proof.
  - clear A...
  - clear B...
Qed. (* spins *)
```

```
Scheme T1_mutind := Induction for T1 Sort Prop
with T2_mutind := Induction for T2 Sort Prop.
Combined Scheme T1_T2_ind from T1_mut_ind, T2_mutind.

Definition A (t : T1) := ...
Definition B (t : T2) := ...

Lemma AB :
  (forall (t : T1), A t) /\ (forall (t : T2), B t).
Proof.
  apply T1_T2_ind...
Qed.
```

Figure 8.2: An example of the pattern we use to prove lemmas by mutual induction, avoiding the hanging `Qed` problem. `T1` and `T2` are mutually inductive types. The top is the original and the bottom is the transformed version.

8.2.3 Finite Maps

In our presentation above, we treat all finite maps uniformly, but in our implementation we use both Coq's `FMap` interface, instantiated with an AVL tree, and our own key-polymorphic finite map using association lists. The reasons for this are threefold.

First, association lists make some termination proofs easier or even automatic, because operations on them are structurally decreasing and compose cleanly with Coq's syntactic termination checking. Second, in the portion of vsa that memoizes deduction rules, we need key-polymorphic finite maps; due to the nature of Coq's module system, `FMap` only supports value polymorphism. Third, the portion of vsa that stores recursive synthesis state violates strict positivity if we use `FMap`: this limitation is quite interesting and suggests that `FMap`'s abstraction is flawed.

```
Record bst (elt : Type) : Type := Bst
  { this   : FMap.Raw.tree elt;
    is_bst : FMap.Raw.bst this }.
```

Figure 8.3: The internal structure of FMap's AVL tree implementation. `bst` stands for "binary search tree". `(FMap.Raw.tree elt)` is a `Type` and `(FMap.Raw.bst this)` is a `Prop`.

`FMap` uses an interface that is common in verified code: the object is a record comprising both the actual data representation and a propositional invariant, which is erased at extraction time. See Figure 8.3 for more concreteness. The invariant essentially states that the object was constructed and manipulated using only functions provided by the interface, which ensures the correctness of theorems about the abstraction. However, physically putting this proposition inside the object representation has unintended consequences, namely that the proposition can run afoul of Coq's strict positivity requirement. See Figure 8.4 for a minimal example of this. Strict positivity states that within the definition of an inductive type `T`, `T` itself cannot appear as the input type to a function. In its absence, one can write expressions that break the soundness of Coq's logic [4].

```
Inductive T :=
| C : FMap.t T -> T.
(* violates strict positivity *)
```

```
Inductive T :=
| C : FMap.Raw.tree T -> T.
(* satisfies strict positivity *)
```

Figure 8.4: A minimal example where storing well-formedness invariants inside FMap violates strict positivity. The left is the entire FMap, including the well-formedness invariant, and the right is just the computational component, i.e. the tree representation.

The stopgap approach that we took to address this—define our own finite map and prove the composition of its properties in the context of a larger system—is clearly not scalable. What would be ideal

is to encode this well-formedness invariant syntactically, so that we could do proofs by inversion on the functions provided by the interface as if they were constructors. Existential types of the kind in practical functional languages, where the implementation type is hidden, would fit naturally here, but as far as we are aware there is no appropriate analogue in Coq: the quantifier `exists` only works for the universe `Prop`, and the `Type` equivalent `sig` allows you to extract the witness. Whether this kind of interface can be represented in Coq—and if not, how best to respond—is an interesting research question. Best practices for verified software engineering are relatively unestablished.

9. Conclusion

In this thesis, we proposed the PbE^2 interaction model, which makes it easier for non-expert developers to create domain-specific synthesis engines. By asking developers to write a corpus of programs in the domain, rather than provide a formal DSL specification, we remove three key barriers in existing frameworks like PROSE: designing DSLs, writing deduction rules, and debugging. These claims are borne out in two small user studies we conducted on college seniors.

We detailed a Coq implementation of PbE^2 as VERSE, which is architected to enable a mechanically verified end-to-end proof of correctness, still in progress. This provides strong formal guarantees both to developers and synthesis experts. Developers using VERSE receive the guarantee that the inferred DSL generalizes their corpus, and that the resulting synthesizer is both sound and complete with respect to the DSL. Experts adding operators to VERSE's library receive the guarantee that if the new operator locally satisfies the formal interface, the entire system will be correct.

There are several promising directions for future work. In increasing order of generality:

First, and most obviously, there is the task of completing the mechanized proof of correctness. Beyond concrete improvements to make VERSE a usable and mature tool, there is interesting research content in enriching the type system of VERSE's general-purpose language (e.g. with inductive types) and removing its synthesis engine's dependence on reified types. When we attempted a first pass at this, we quickly ran up against the predicativity of Coq's type universes; doing this successfully would be a challenging exercise in dependent types, where Coq's proof machinery is lacking, and metaprogramming, which is only supported through external projects like MetaCoq [73].

Second, there are several dimensions along which we could improve VERSE's inference algorithm. As inference is just synthesis where there is a unique correct solution—and our correctness criterion is actually a tradeoff space between expressivity and performance—we can cast the problem as bona fide DSL synthesis. In particular, the output would be a concise representation of the DSL space that can be traversed by an arbitrary (perhaps machine-learned) ranking function. This architecture would allow us to change the tradeoff between DSL expressivity and synthesis performance without modifying the under-

lying inference algorithm.

Third, we could take a cue from Wang et al. [86] and synthesize our deduction rules, rather than require an expert to write them. As VERSE formally specifies the behavior of deduction rules in terms of forward semantics, this is a straightforward synthesis problem (albeit complicated if we want to synthesize Coq programs and proofs).

Fourth, we could investigate PbE^2 from a human–computer interaction perspective. This would entail both a more comprehensive user study and improvements to PbE^2's design, in particular making DSL inference mixed initiative. A naive approach would pose membership queries to the developer, but it might be more effective to ask higher-level questions, like whether an appearance of the number 0 in a program should be hard-coded in the DSL or generalized to a natural number or integer.

The usability of synthesis tools is improving faster for end users than it is for developers, and narrow deployment limits the benefits that synthesis technology brings to people in the wild. We hope that PbE^2 serves as an example of how we can use insights from interaction models for users to improve the experience of developers.

References

[1] (Accessed April 10, 2020b). Issues · coq/coq · GitHub. https://github.com/coq/coq/issues?q=label%3A%22kind%3A+inconsistency%22+is%3Aclosed.

[2] (Accessed April 19, 2020). CPDT: More Dep. http://adam.chlipala.net/cpdt/html/Cpdt.MoreDep.html.

[3] (Accessed April 7, 2020a). Coq Kernel. https://github.com/coq/coq/tree/master/kernel.

[4] (Accessed April 8, 2020). CPDT: Inductive Types. http://adam.chlipala.net/cpdt/html/InductiveTypes.html.

[5] (Accessed December 11, 2019). Issues · Z3Prover/z3 · GitHub. https://github.com/Z3Prover/z3/issues?q=label%3Abug+is%3Aclosed.

[6] (Accessed December 11, 2019). Snippet from PROSE DSL authoring tutorial. https://github.com/microsoft/prose/blob/b6efeec2479977b6184e11af749c5071cf94ffe0/DslAuthoringTutorial/part1c/ProseTutorial/synthesis/WitnessFunctions.cs#L72.

[7] (Accessed December 11, 2019). Z3Prover/z3: The Z3 Theorem Prover. https://github.com/Z3Prover/z3. Lines of code in `z3/src` counted using https://github.com/cgag/loc.

[8] (Archived April 4, 2020). Microsoft PROSE SDK – Standard Concepts. https://web.archive.org/web/20200404214152/https://microsoft.github.io/prose/documentation/prose/concepts/.

[9] (Archived March 27, 2020). Communication with PROSE developers. https://github.com/microsoft/prose/issues/42#issuecomment-605326085.

[10] (Archived March 30, 2020). Microsoft PROSE SDK. http://web.archive.org/web/20200330173912/https://microsoft.github.io/prose/.

[11] (Archived March 30, 2020). xkcd: Is It Worth the Time? http://web.archive.org/web/20200330174015/https://xkcd.com/1205/.

[12] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I. J., Harp, A., Irving, G., Isard, M., Jia, Y., Józefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D. G.,

Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P. A., Vanhoucke, V., Vasudevan, V., Viégas, F. B., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., & Zheng, X. (2016). Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *CoRR*, abs/1603.04467.

[13] Alur, R., Bodik, R., Juniwal, G., Martin, M. M. K., Raghothaman, M., Seshia, S. A., Singh, R., Solar-Lezama, A., Torlak, E., & Udupa, A. (2013). Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design* (pp. 1–8).

[14] Alur, R., Cerný, P., & Radhakrishna, A. (2015). Synthesis through unification. *CoRR*, abs/1505.05868.

[15] Anand, A., Appel, A. W., Morrisett, G., Paraskevopoulou, Z., Pollack, R., Bélanger, O. S., Sozeau, M., & Weaver, M. (2016). Certicoq : A verified compiler for coq.

[16] Angluin, D. (1987). Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2), 87–106.

[17] Angluin, D. (1988). Queries and concept learning. *Machine Learning*, 2(4), 319–342.

[18] Balog, M., Gaunt, A. L., Brockschmidt, M., Nowozin, S., & Tarlow, D. (2016). Deepcoder: Learning to write programs. *CoRR*, abs/1611.01989.

[19] Baydin, A. G., Pearlmutter, B. A., & Radul, A. A. (2015). Automatic differentiation in machine learning: a survey. *CoRR*, abs/1502.05767.

[20] Biermann, A. W. (1978). The inference of regular lisp programs from examples. *IEEE Transactions on Systems, Man, and Cybernetics*, 8(8), 585–600.

[21] Blumer, A., Ehrenfeucht, A., Haussler, D., & Warmuth, M. K. (1987). Occam's razor. *Information Processing Letters*, 24(6), 377–380.

[22] Bodik, R. (October 10, 2019). Rethinking software 50 operations at a time. https://video.seas.harvard.edu/media/CS+Colloquium+Rastislav+Bodik+2019-10-10/1_h9rmblxm/13151381. Presentation in Harvard CS Colloquium Series.

[23] Bodik, R. & Torlak, E. (2012b). Synthesizing programs with constraint solvers. https://www.cs.umd.edu/class/spring2013/cmsc631/lectures/synthesis.pdf. Computer Aided Verification (CAV) 2012 Invited Tutorial.

[24] Bodik, R. & Torlak, E. (Fall 2012a). Cs294: Program synthesis for everyone. https://homes.cs.washington.edu/~bodik/ucb/cs294fa12.html.

[25] Buchi, J. R. & Landweber, L. H. (1969). Solving sequential conditions by finite-state strategies. *Transactions of the American Mathematical Society*, 138, 295–311.

[26] Castéran, P. & Bertot, Y. (2004). *Interactive theorem proving and program development. Coq'Art: The Calculus of inductive constructions*. Texts in Theoretical Computer Science. Springer Verlag.

[27] de Moura, L. & Bjørner, N. (2008). Z3: An efficient smt solver. In C. R. Ramakrishnan & J. Rehof (Eds.), *Tools and Algorithms for the Construction and Analysis of Systems* (pp. 337–340). Berlin, Heidelberg: Springer Berlin Heidelberg.

[28] Delaware, B., Claudel, C., Gross, J., & Chlipala, A. (2015). Fiat: Deductive synthesis of abstract data types in a proof assistant. *Acm Sigplan Notices*, 50(1), 689–700.

[29] Eén, N. & Sörensson, N. (2004). An extensible sat-solver. In E. Giunchiglia & A. Tacchella (Eds.), *Theory and Applications of Satisfiability Testing* (pp. 502–518). Berlin, Heidelberg: Springer Berlin Heidelberg.

[30] Fantl, J. (2017). Knowledge how. In E. N. Zalta (Ed.), *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, fall 2017 edition.

[31] Frankle, J., Osera, P.-M., Walker, D., & Zdancewic, S. (2016). Example-directed synthesis: a type-theoretic interpretation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on principles of programming languages*, volume 51 of *POPL 2016* (pp. 802–815).: ACM.

[32] Gold, E. M. (1967). Language identification in the limit. *Information and Control*, 10(5), 447–474.

[33] Gross, J., Erbsen, A., & Chlipala, A. (2018). Reification by parametricity. In J. Avigad & A. Mahboubi (Eds.), *Interactive Theorem Proving* (pp. 289–305). Cham: Springer International Publishing.

[34] Gu, R., Shao, Z., Chen, H., Kim, J., Koenig, J., Wu, X., Sjoberg, V., & Costanzo, D. (2019). *Building Certified Concurrent OS Kernels.(CertiKOS for building verified concurrent operating system kernels)(Research Highlights)(Technical report)*. Technical Report 10.

[35] Gulwani, S. (2011). Automating string processing in spreadsheets using input-output examples. In *PoPL'11, January 26-28, 2011, Austin, Texas, USA*.

[36] Gulwani, S., Polozov, O., & Singh, R. (2017). Program synthesis. *Foundations and Trends in Programming Languages*, 4(1-2), 1–119.

[37] Hardy, S. (1974). Automatic induction of lisp functions. In *Proceedings of the 1st Summer Conference on Artificial Intelligence and Simulation of Behaviour*, AISB'74 (pp. 50–62). NLD: IOS Press.

[38] Hur, C.-K., Neis, G., Dreyer, D., & Vafeiadis, V. (2013). The power of parameterization in coinductive proof. *ACM SIGPLAN Notices*, 48(1), 193–206.

[39] Itzhaky, S., Singh, R., Solar-Lezama, A., Yessenov, K., Lu, Y., Leiserson, C., & Chowdhury, R. (2016). Deriving divide-and-conquer dynamic programming algorithms using solver-aided transformations. *ACM SIGPLAN Notices*, 51(10), 145–164.

[40] Jha, S., Gulwani, S., Seshia, S. A., & Tiwari, A. (2010). Oracle-guided component-based program synthesis. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 1 (pp. 215–224).: IEEE.

[41] Knobe, B. & Knobe, K. (1976). A method for inferring context-free grammars. *Information and Control*, 31(2), 129–146.

[42] Langley, P. & Stromsten, S. (2000). Learning context-free grammars with a simplicity bias. In R. López de Mántaras & E. Plaza (Eds.), *Machine Learning: ECML 2000* (pp. 220–228). Berlin, Heidelberg: Springer Berlin Heidelberg.

[43] Lau, T. (2009). Why programming by demonstration systems fail: Lessons learned for usable ai. *AI Magazine*, 30(4), 65–67.

[44] Le, V. & Gulwani, S. (2014). Flashextract: A framework for data extraction by examples. *ACM SIGPLAN Notices*, 49.

[45] Le, V., Perelman, D., Polozov, O., Raza, M., Udupa, A., & Gulwani, S. (2017). Interactive program synthesis. *CoRR*, abs/1703.03539.

[46] Lee, W., Heo, K., Alur, R., & Naik, M. (2018). Accelerating search-based program synthesis using learned probabilistic models. *ACM SIGPLAN Notices*, 53(4), 436–449.

[47] Leroy, X. (2009a). Formal verification of a realistic compiler. *Communications of the ACM*, 52(7), 107–115.

[48] Leroy, X. (2009b). A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4), 363–446.

[49] Lindblad, F. & Benke, M. (2006). A tool for automated theorem proving in agda. In *Proceedings of the 2004 International Conference on Types for Proofs and Programs*, TYPES'04 (pp. 154–169). Berlin, Heidelberg: Springer-Verlag.

[50] Mayer, M., Soares, G., Grechkin, M., Le, V., Marron, M., Polozov, O., Singh, R., Zorn, B., & Gulwani, S. (2015). User interaction models for disambiguation in programming by example. In *Proceedings of the 28th Annual ACM Symposium on user interface software & technology*, UIST '15 (pp. 291–301).: ACM.

[51] McBride, C. (2002). Elimination with a motive. In P. Callaghan, Z. Luo, J. McKinna, R. Pollack, & R. Pollack (Eds.), *Types for Proofs and Programs* (pp. 197–216). Berlin, Heidelberg: Springer Berlin Heidelberg.

[52] Mitchell, T. M. (1982). Generalization as search. *Artificial Intelligence*, 18(2), 203–226.

[53] Mullen, E., Pernsteiner, S., Wilcox, J. R., Tatlock, Z., & Grossman, D. (2018). undefineduf: Minimizing the coq extraction tcb. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018 (pp. 172–185). New York, NY, USA: Association for Computing Machinery.

[54] Myers, B. A., Ko, A. J., & Burnett, M. M. (2006). Invited research overview: End-user programming. In *CHI '06 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '06 (pp. 75–80). New York, NY, USA: ACM.

[55] Nevill-Manning, C. G. & Witten, I. H. (1997). Identifying hierarchical structure in sequences: A linear-time algorithm. *CoRR*, cs.AI/9709102.

[56] Nipkow, T., Wenzel, M., & Paulson, L. C. (2002). *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Berlin, Heidelberg: Springer-Verlag.

[57] Oates, T., Armstrong, T., Harris, J., & Nejman, M. (2003). Leveraging lexical semantics to infer context-free grammars. In *Proceedings of the 2003rd European Conference on Learning Context-Free Grammars*, ECMLLCFG'03 (pp. 65–76). Zagreb, HRV: Ruder Boskovic Institute.

[58] Osera, P.-M. & Zdancewic, S. (2015). Type-and-example-directed program synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15 (pp. 619–630). New York, NY, USA: ACM.

[59] Padhi, S., Millstein, T., Nori, A., & Sharma, R. (2019). Overfitting in synthesis: Theory and practice (extended version). *arXiv.org*.

[60] Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., & Lerer, A. (2017). Automatic differentiation in pytorch.

[61] Peleg, H., Shoham, S., & Yahav, E. (2017). Programming not only by example. *arXiv.org*.

[62] Polikarpova, N. & Sergey, I. (2018). Structuring the synthesis of heap-manipulating programs - extended version. *arXiv.org*, 3(POPL).

[63] Polikarpova, N. & Solar-Lezama, A. (2015). Program synthesis from polymorphic refinement types. *CoRR*, abs/1510.08419.

[64] Polosukhin, I. & Skidanov, A. (2018). Neural program search: Solving programming tasks from description and examples. *CoRR*, abs/1802.04335.

[65] Polozov, A. & Gulwani, S. (2015a). Flashmeta: A framework for inductive program synthesis. In *ACM SIGPLAN conference on Systems, Programming, Languages and Applications: Software for Humanity (SPLASH) 2015*.

[66] Polozov, O. & Gulwani, S. (2015b). Microsoft prose sdk: A framework for inductive program synthesis.

[67] Sakakibara, Y. (1990). Learning context-free grammars from structural data in polynomial time. *Theoretical Computer Science*, 76(2), 223 – 242.

[68] Scherer, G. (2017). Search for program structure. In *2nd Summit on Advances in Programming Languages (SNAPL 2017)*.

[69] Shamir, E. (1962). A remark on discovery algorithms for grammars. *Information and Control*, 5(3), 246–251.

[70] Smith, D. C. (1975). *PYGMALION: A Creative Programming Environment*. Technical report.

[71] Solar-Lezama, A. (2008). *Program Synthesis by Sketching*. PhD thesis, Berkeley, CA, USA. AAI3353225.

[72] Solar-Lezama, A., Tancau, L., Bodik, R., Seshia, S., & Saraswat, V. (2006). Combinatorial sketching for finite programs. *ACM SIGPLAN Notices*, 41(11), 404–415.

[73] Sozeau, M., Anand, A., Boulier, S., Cohen, C., Forster, Y., Kunze, F., Malecha, G., Tabareau, N., & Winterhalter, T. (2019a). The metacoq project.

[74] Sozeau, M., Boulier, S., Forster, Y., Tabareau, N., & Winterhalter, T. (2019b). Coq coq correct! verification of type checking and erasure for coq, in coq. *Proc. ACM Program. Lang.*, 4(POPL).

[75] Srivastava, S., Gulwani, S., & Foster, J. S. (2010). From program verification to program synthesis. *ACM SIGPLAN Notices*, 45(1), 313.

[76] Stevenson, A. & Cordy, J. R. (2014). A survey of grammatical inference in software engineering. *Science of Computer Programming*, 96(4), 444–459.

[77] Streicher, T. (1993). Investigations into intensional type theory.

[78] Summers, P. (1977). A methodology for lisp program construction from examples. *Journal of the ACM (JACM)*, 24(1), 161–175.

[79] Tanzer, G. & Wendland, A. (2019). VERSE: Programming "Programming by Example" by Example. https://github.com/gtanzer/verse-hci.

[80] Torlak, E. & Bodik, R. (2013). Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013 (pp. 135–152). New York, NY, USA: ACM.

[81] Udupa, A., Raghavan, A., Deshmukh, J. V., Mador-Haim, S., Martin, M. M., & Alur, R. (2013). Transit: Specifying protocols with concolic snippets. *SIGPLAN Not.*, 48(6), 287–296.

[82] van Tonder, R. & Le Goues, C. (2018). Static automated program repair for heap properties. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18 (pp. 151–162). New York, NY, USA: Association for Computing Machinery.

[83] Črepinšek, M., Mernik, M., Bryant, B. R., Javed, F., & Sprague, A. (2005). Inferring context-free grammars for domain-specific languages. *Electronic Notes in Theoretical Computer Science*, 141(4), 99–116.

[84] Vijayakumar, A. J., Mohta, A., Polozov, O., Batra, D., Jain, P., & Gulwani, S. (2018). Neural-guided deductive search for real-time program synthesis from examples. *CoRR*, abs/1804.01186.

[85] Waldinger, R. J. & Lee, R. C. T. (1969). Prow: A step toward automatic program writing. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence*, IJCAI'69 (pp. 241–252). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

[86] Wang, X., Anderson, G., Dillig, I., & Mcmillan, K. (2018). Learning abstractions for program synthesis. *arXiv.org*.

[87] Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3), 229–256.

[88] Wintersteiger, C. M., Hamadi, Y., & Moura, L. (2009). A concurrent portfolio approach to smt solving. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV '09 (pp. 715–720). Berlin, Heidelberg: Springer-Verlag.

[89] Wolfman, S., Lau, T., Domingos, P., & Weld, D. (2001). Mixed initiative interfaces for learning tasks: Smartedit talks back. In *Proceedings of the 6th international conference on intelligent user interfaces*, IUI '01 (pp. 167–174).: ACM.

[90] Zhang, L., Rosenblatt, G., Fetaya, E., Liao, R., Byrd, W. E., Might, M., Urtasun, R., & Zemel, R. S. (2018). Neural guided constraint logic programming for program synthesis. *CoRR*, abs/1809.02840.